Introduction to Data-Flow and Control-Flow Analyses

Y.N. Srikant

(Formerly) Professor
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

ACM Winter School on Design, Implementation and Verification of Computer Systems
January 3-16, 2022.

Outline of the Lecture

- Introduction
- Examples of data-flow analyses
- Fundamentals of control-flow analysis
- Interprocedural data-flow analysis
- Points-to analysis

Data-flow analysis

- These are techniques that derive information about the flow of data along program execution paths.
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, ..., p_n$ in the program that occur one after another.
- In general, there are infinite number of paths through a program and there is no bound on the length of a path.

Data-flow analysis

- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts.
- No analysis is necessarily a perfect representation of the state.
- DFAs collect information necessary to perform program optimizations.

Uses of Data-flow Analysis

- Program debugging
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*.
- Program optimizations
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.

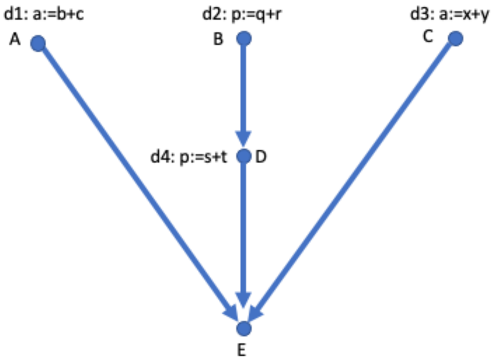
Data-Flow Analysis Schema(1)

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point.
- The set of all possible data-flow values is the *domain* for the application under consideration.
  - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of of definitions in the program.
  - A particular data-flow value is a set of definitions.
- $IN[B]$ and $OUT[B]$: data-flow values *before* and *after* each basic block B.
- The *data-flow problem* is to find a solution to a set of constraints on $IN[B]$ and $OUT[B]$, for all basic blocks B.

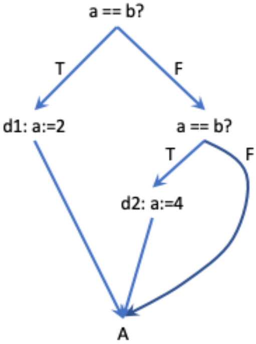
Data-Flow Analysis Schema (2)

- Two kinds of constraints
  - Those based on the semantics of statements (*transfer functions*).
  - Those based on flow of control.
- A DFA schema consists of
  - A control-flow graph.
  - A direction of data-flow (forward or backward).
  - A set of data-flow values.
  - A confluence operator (usually set union or intersection).
  - Transfer functions for each block.
- We always compute *safe* estimates of data-flow values.
- A decision or estimate is *safe* or *conservative*, if it never leads to a change in what the program computes (before and after using the estimated value).
- These safe values may be either subsets or supersets of actual values, based on the application.

The Reaching Definitions Problem(1)



d1, d3, and d4 *reach* E.
d2 reaches only D. d4 *kills* d2.



Both d1 and d2 *reach* A. This is safe. It is safe to assume that a definition reaches a point even if it does not.

The Reaching Definitions Problem(2)

- We *kill* a definition of a variable *a*, if between two points along the path, there is an assignment to *a*.
- A definition *d* *reaches* a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not *killed* along that path.
- We compute supersets of definitions as *safe* values.
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both `a=2` and `a=4` reach the point after the complete if-then-else statement, even though the statement `a=4` is not reached by control flow:

```
if (a==b) a=2; else if (a==b) a=4;
```

The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$
\begin{aligned}
IN[B] &= \bigcup_{P \text{ is a predecessor of } B} OUT[P] \\
OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]) \\
IN[B] &= \phi, \textit{ for all } B \ (\textit{initialization only})
\end{aligned}
$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set.
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\bigcup$.
  - Direction of flow does not imply traversing the basic blocks in a particular order.
  - The final result does not depend on the order of traversal of the basic blocks.

The Reaching Definitions Problem (4)

- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
  - If a variable x has two or more defintions in a basic block, then only the last definition of x is downwards exposed; all others are not visible outside the block
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B
  - If a variable x has a definition d_i in a basic block, then d_i kills all the definitions of the variable x in the program, except d_i

Reaching Definitions Analysis: GEN and KILL

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

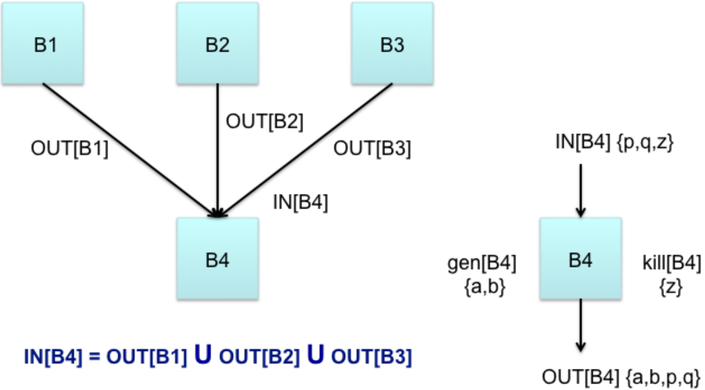
d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

B

Set of all definitions = {d1,d2,d3,d4,d5,d6,d7,d8,d9,10}

GEN[B] = {d2,d3,d4}
KILL[B] = {d4,d9,d5,d10,d1}

Reaching Definitions Analysis: DF Equations



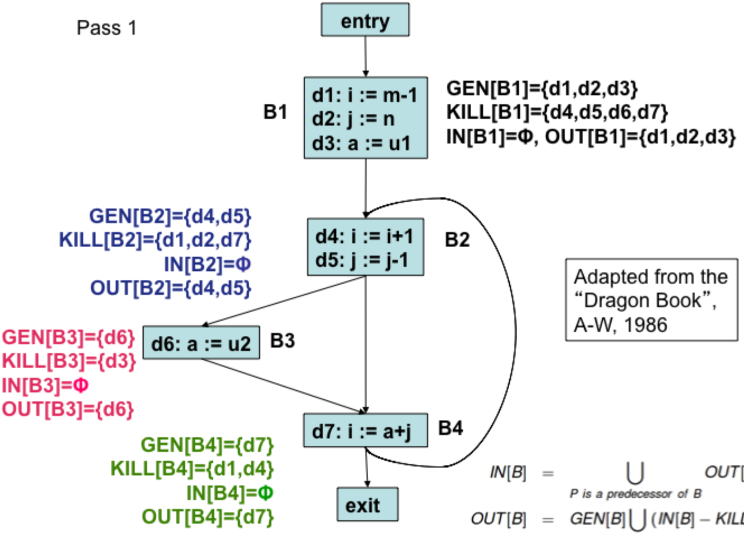
IN[B4] = OUT[B1] U OUT[B2] U OUT[B3]

$$
\begin{aligned}
IN[B] &= \bigcup_{P \text{ is a predecessor of } B} OUT[P] \\
OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B])
\end{aligned}
$$

OUT[B4] = gen[B4] U (IN[B4] – kill[B4])

Reaching Definitions Analysis: An Example - Pass 1

Pass 1



Adapted from the "Dragon Book", A-W, 1986

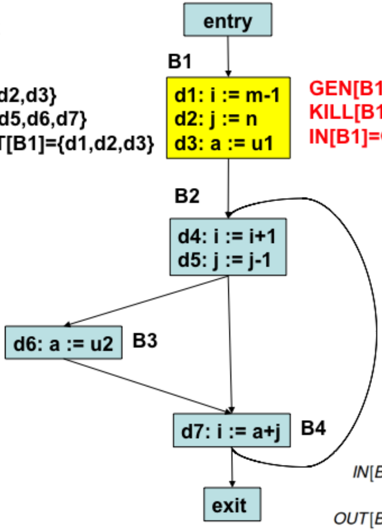
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Pass 2.1

Pass 2

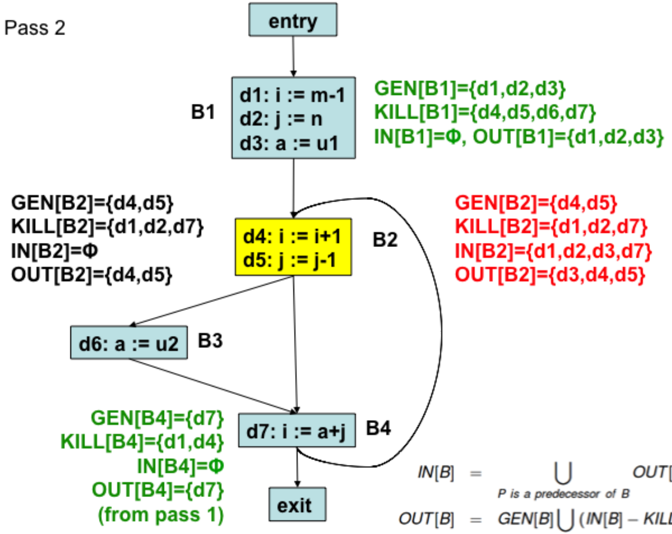
GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

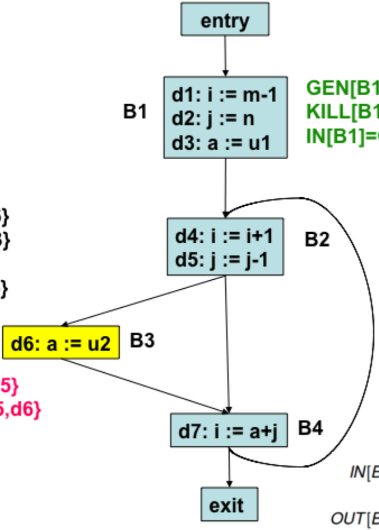


$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Pass 2.3

Pass 2



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

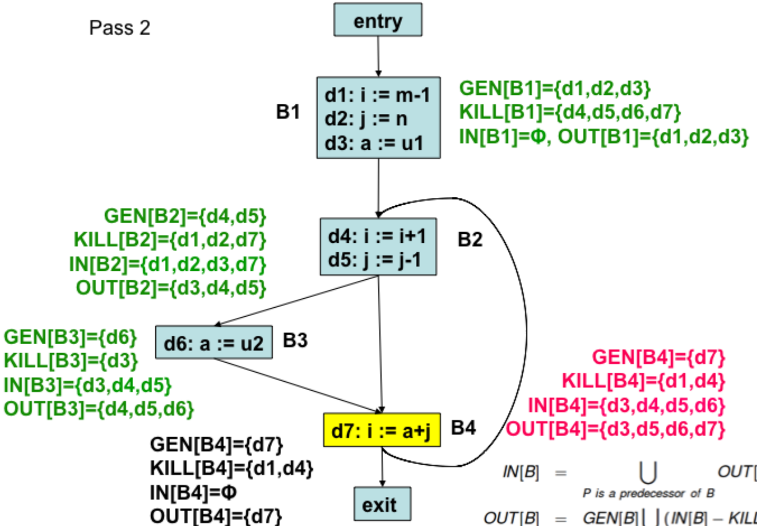
GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]=Φ
OUT[B3]={d6}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5}
OUT[B3]={d4,d5,d6}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Pass 2

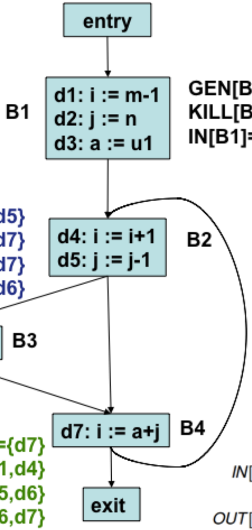


$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Final

Final



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d5,d6,d7}
OUT[B2]={d3,d4,d5,d6}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5,d6}
OUT[B3]={d4,d5,d6}

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]={d3,d4,d5,d6}
OUT[B4]={d3,d5,d6,d7}

Adapted from the "Dragon Book", A-W, 1986

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

An Iterative Algorithm for Computing Reaching Def.

for each block B do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
$change = true$;
while $change$ do { $change = false$;
for each block B do {

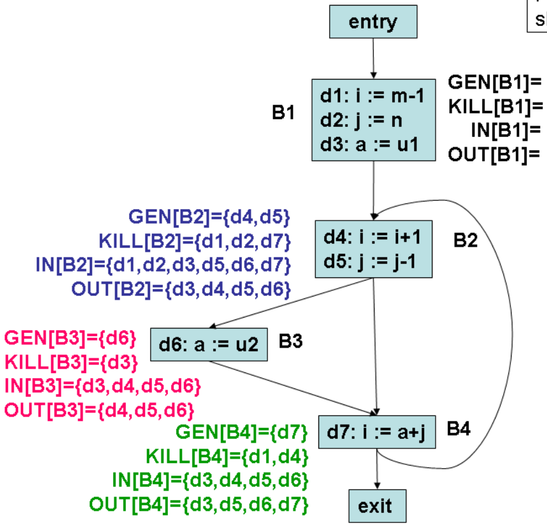
$$
\begin{aligned}
IN[B] &= \bigcup_{P \text{ a predecessor of } B} OUT[P]; \\
oldout &= OUT[B]; \\
OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]);
\end{aligned}
$$

if ($OUT[B] \neq oldout$) $change = true$;
}
}

- *GEN*, *KILL*, *IN*, and *OUT* are all represented as bit vectors with one bit for each definition in the flow graph

Reaching Definitions: Bit Vector Representation

Final dataflow value sets shown in bit vector format



	d1	d2	d3	d4	d5	d6	d7
GEN[B1]=	1	1	1	0	0	0	0
KILL[B1]=	0	0	0	1	1	1	1
IN[B1]=	0	0	0	0	0	0	0
OUT[B1]=	1	1	1	0	0	0	0

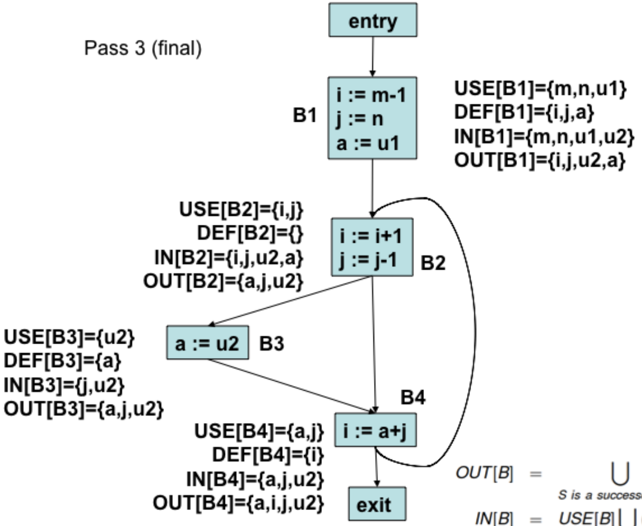
Adapted from the “Dragon Book”, A-W, 1986

Live Variable Analysis (1)

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p.
- Sets of variables constitute the domain of data-flow values.
- Backward flow problem, with confluence operator $\bigcup$.
- $IN[B]$ is the set of variables live at the beginning of B.
- $OUT[B]$ is the set of variables live just after B.
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B.
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable.

$$
\begin{aligned}
OUT[B] &= \bigcup_{S \text{ is a successor of } B} IN[S] \\
IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B]) \\
IN[B] &= \phi, \textit{ for all } B \textit{ (initialization only)}
\end{aligned}
$$

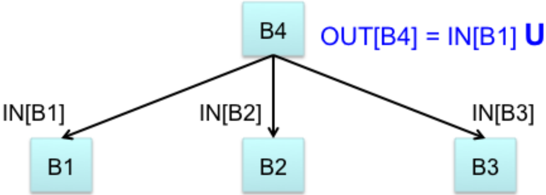
Live Variable Analysis: An Example



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

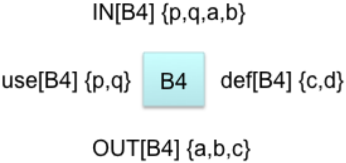
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis (2)



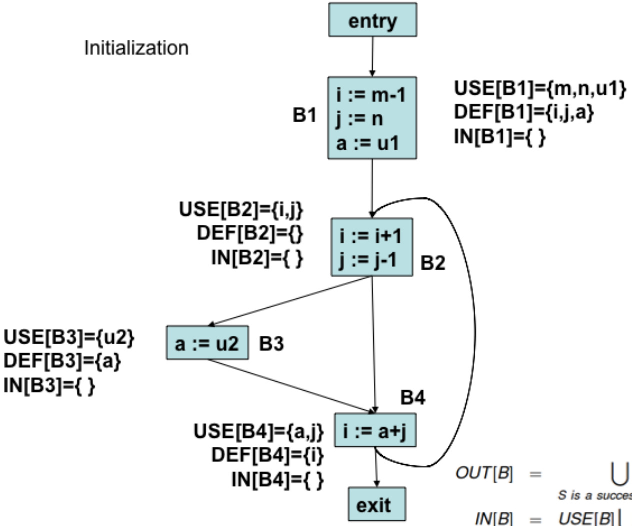
OUT[B4] = IN[B1] **U** IN[B2] **U** IN[B3]

$$
\begin{aligned}
OUT[B] &= \bigcup_{S \text{ is a successor of } B} IN[S] \\
IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B]) \\
IN[B] &= \phi, \text{for all } B \text{ (initialization only)}
\end{aligned}
$$



IN[B4] = use[B4] **U** (OUT[B4] – def[B4])

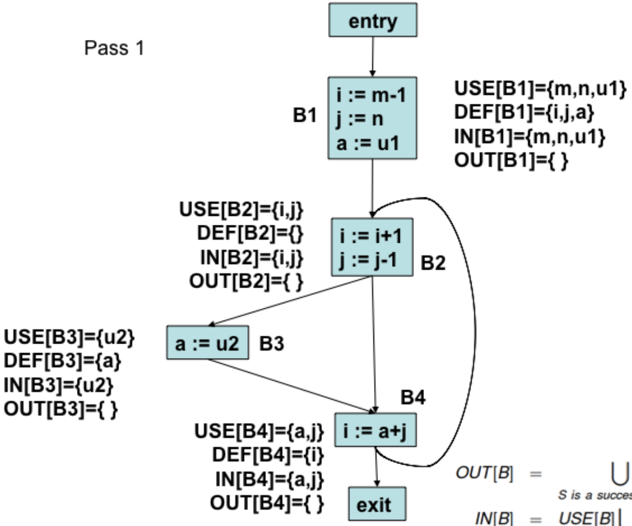
Live Variable Analysis: An Example - Init



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

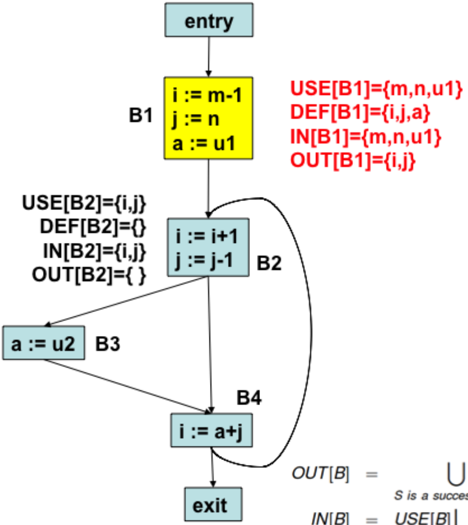
Live Variable Analysis: An Example - Pass 1



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

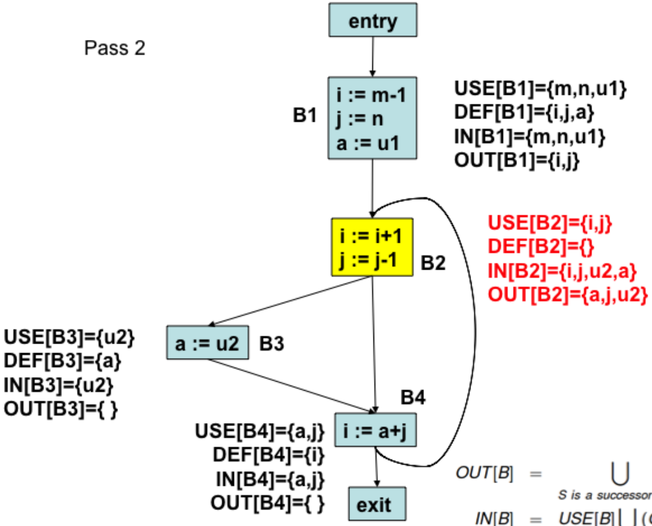
Pass 2



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

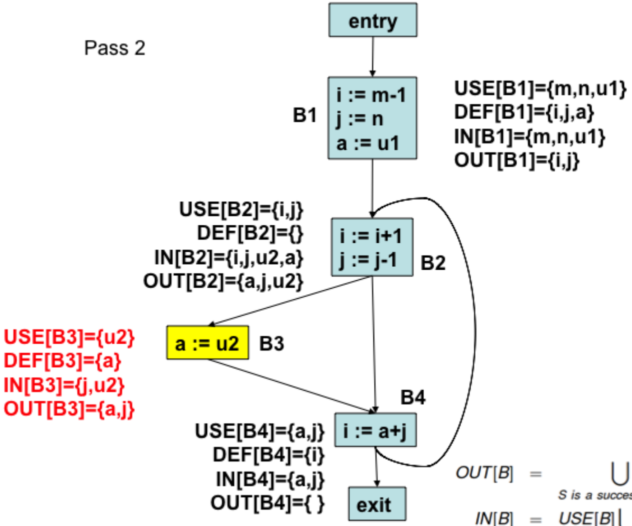
Live Variable Analysis: An Example - Pass 2.2



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

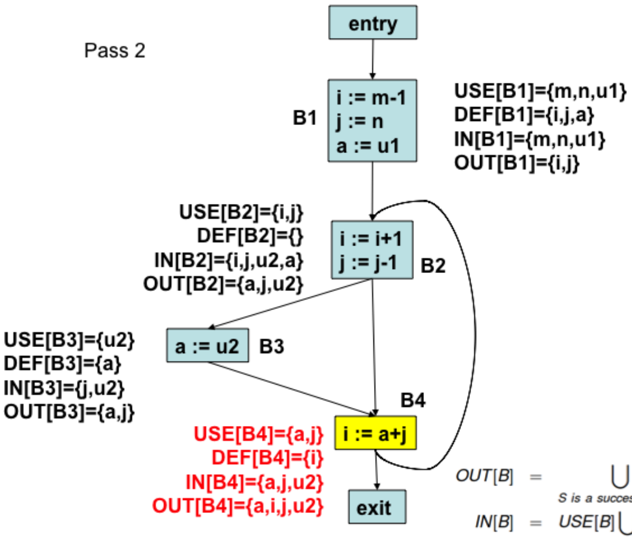
Pass 2



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

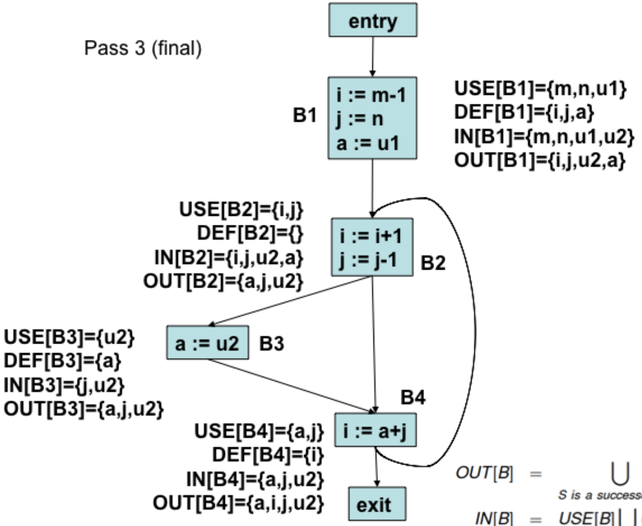
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis: An Example - Pass 2.4



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Control-Flow Analysis

Introduction

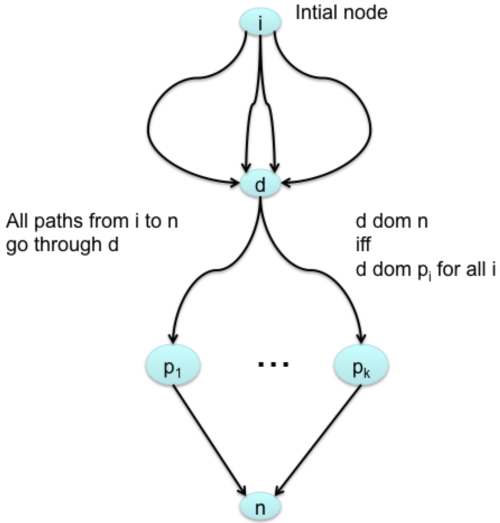
Control-flow analysis (CFA) helps us to understand the structure of control-flow graphs (CFG).

- To determine the loop structure of CFGs.
- To compute dominators - useful for code motion.
- To compute dominance frontiers - useful for the construction of the static single assignment form (SSA).
- To compute control dependence - needed in parallelization.

Dominators

- We say that a node d in a flow graph *dominates* node n, written d *dom* n, if every path from the initial node of the flow graph to n goes through d.
- Initial node is the root, and each node dominates only its descendents in the dominator tree (including itself).
- Principle of the dominator algorithm
  - If $p_1, p_2, ..., p_k$, are all the predecessors of n, and $d \neq n$, then d *dom* n, iff d *dom* p_i for each i.

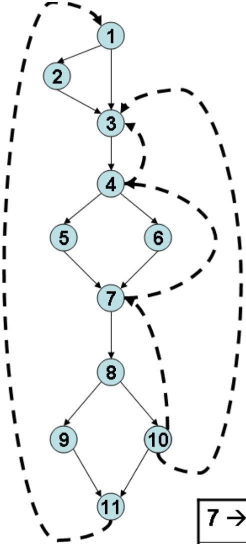
Dominator Algorithm Principle



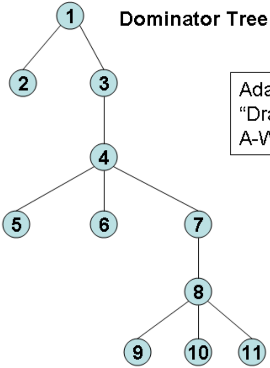
Dominators and Natural Loops

- Edges whose heads dominate their tails are called *back edges* ($a \rightarrow b:\ b = head,\ a = tail$)
- Given a back edge $n \rightarrow d$
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop
    - A single entry point to the loop that dominates all nodes in the loop
    - At least one path back to the header exists (so that the loop can be iterated)

Dominators, Back Edges, and Natural Loops(1)



Flow Graph



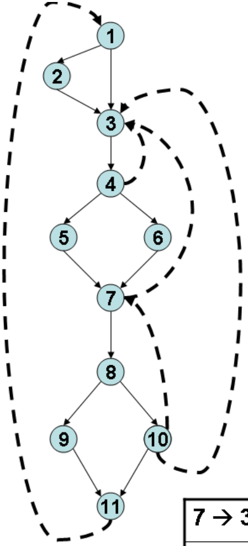
Dominator Tree

Adapted from the "Dragon Book", A-W, 1986

Back edges and their natural loops

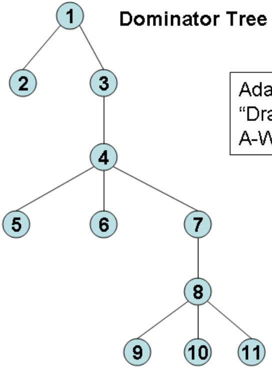
7 → 4	10→7	4→3	10→3	11→1
{4,5,6,7,8, 10}	{7,8,10}	{3,4,5,6,7, 8,10}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Dominators, Back Edges, and Natural Loops(2)



Flow Graph

Dominator Tree



Adapted from the "Dragon Book", A-W, 1986

Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8, 10}	{7,8,10}	{3,4}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Interprocedural Data-Flow Analysis

Introduction

Intraprocedural DFA

- Performed on one procedure at a time.
- Assumes that a procedure invocation may alter all the "visible" variables.
- Imprecise, conservative, but simple.

Interprocedural DFA

- Operates across an entire program and is more complex.
- Makes information flow from caller to callee and vice-versa.
- Procedure inlining is a simple method to enable such information flow.
  - Applicable only if target of a call is known.
  - Increases memory footprint.
- Applications
  - Converting virtual method calls to static method calls.
  - Checking array bounds.
  - Better constant propagation.

Interprocedural DFA

Context-Insensitive IDFA

- Treat each call and return as goto operations.
- Create a super control flow graph .
- Apply standard analysis on the super CFG.
- Simple, but imprecise.
  - A function is analyzed as a common entity for all its calls.
  - Only input-output behaviour of a function is abstracted out.

Context-Sensitive IDFA

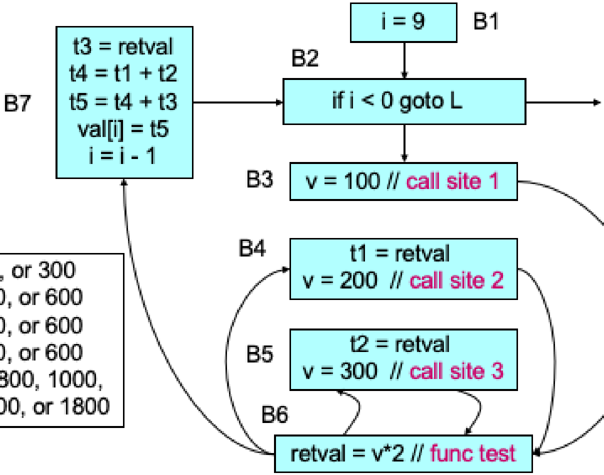
- Calling context (call site) determines the effect of the call.
- In the example (next slide), function test is invoked with a constant in each of the call sites, but the value of the constant is context-dependent.

IDFA - Example Program

```
i = 9;
while (i >= 0) {
      t1 = test(100); // call site 1
      t2 = test(200); // call site 2
      t3 = test(300); // call site 3
      val[i--] = t1 + t2 + t3;
}
int test (int v) {
    return (v*2);
}
```

Context-insensitive IDFA: function test is deemed to return 200, 400, or 600 for any of the three calls.
Context-sensitive IDFA: with interprocedural constant propagation, t1, t2, and t3 will get precisely 200, 400, and 600 respectively.

Super CFG and Context-Insensitive IDFA



v (B6): 100, 200, or 300
t1 (B4): 200, 400, or 600
t2 (B5): 200, 400, or 600
t3 (B7): 200, 400, or 600
val[i] (B7): 600, 800, 1000, 1200, 1400, 1600, or 1800

Cloning and Context-Sensitive IDFA

Simple, context-insensitive analysis is enough on the cloned call graph

```
i = 9;
while (i >= 0) {
    t1 = f1 (100); // call site c1
    t2 = f2 (200); // call site c2
    t3 = f3 (300); // call site c3
    val[i--] = t1 + t2 + t3;
}
int f1 (int v) {
    return test1 (v); // call site c4.1
}
int test1 (int v) {
  return (v*2);
}
```

```
int f2 (int v) {
    return test2 (v); // call site c4.2
}
int test2 (int v) {
   return (v*2);
}
int f3 (int v) {
    return test3 (v); // call site c4.3
}
int test3 (int v) {
   return (v*2);
}
```

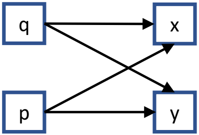
Recursive programs cannot be handled

Points-To Analysis

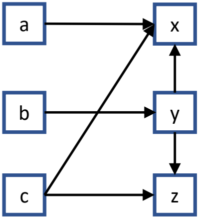
Introduction

Points-to analysis discovers what each pointer variable and heap reference points to.

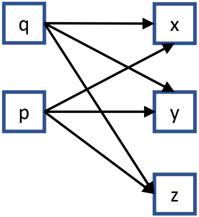
```
q := &x
q := &y
p := q
```




```
a := &x
b := &y
if p then
    y := &z
else
    y := &x
fi
c := *y
```

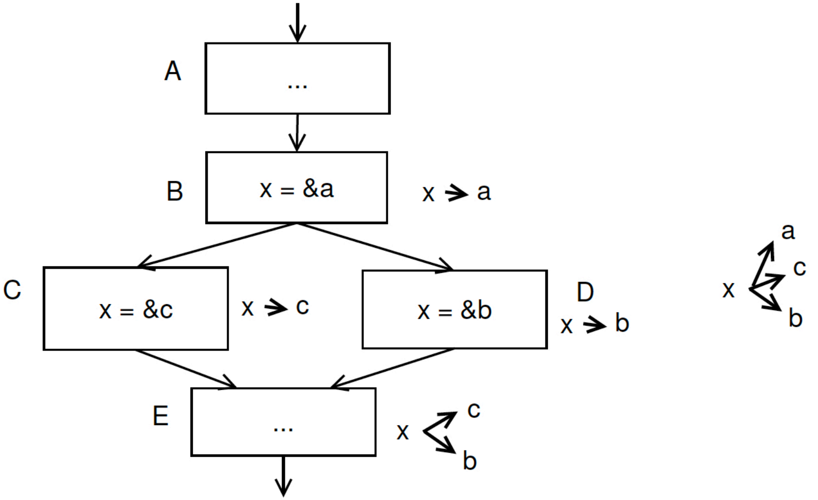



```
q := &x
q := &y
p := q
q := &z
```



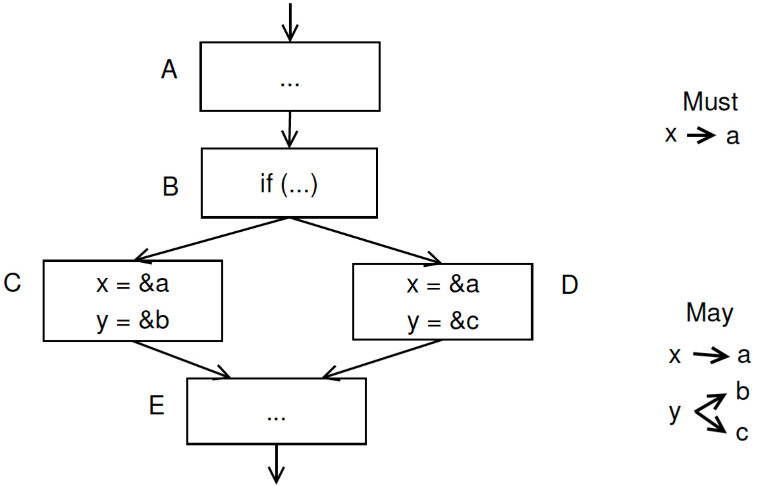
These are flow-insensitive analyses.

Flow-Insensitive vs. Flow-Sensitive Analysis



courtesy: Subhajit Roy

May vs. Must Analysis



courtesy: Subhajit Roy