Module 09: CS31003: Compilers

Data Flow Analysis

Partha Pratim Das & Pralay Mitra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in ; pralay@cse.iitkgp.ac.in

October 05, 2021

Module Objectives

- Understanding Data Flow Analysis (DFA) to estimate various data propagation entities in programs statically
- Understanding DFA formulation with forward / backward flow and inclusive / exclusive confluence
- Understanding formulation for various DFA solutions for Reaching Definitions, Available Expressions, Live Variables, Def-Use Chains, Copy Propagation etc.
- Understanding use of DFA in global optimization

Module Outline

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 $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n-1$, either
  [1] p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  [2] p_i is the end of some block and p_{i+1} is the beginning of a successor block
- In general, there is an *infinite number of paths* through a program and there is *no bound on the length of a path*
- 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*

Path Examples


```
    p0
100: n = 5
    p1
101: i = 0
    p2
102: if i < n goto 106
    p3
103: goto 124
    p4
104: i = i + 1
    p5
105: goto 102
    p6
106: t4 = i << 2
    p7
107: t5 = a[t4]
    p8
```

```
    p8
108: t6 = i << 2
    p9
109: t7 = b[t6]
    p10
110: if t5 >= t7 goto 120
    p11
111: t8 = i << 2
    p12
112: t9 = c + t8
    p13
113: t10 = i << 2
    p14
114: t11 = a[t10]
    p15
115: t12 = i << 2
    p16
```

```
    p16
116: t13 = b[t12]
    p17
117: t14 = t11 * t13
    p18
118: *t9 = t14
    p19
119: goto 104
    p20
120: t15 = i << 2
    p21
121: t16 = c + t15
    p22
122: *t16 = 0
    p23
123: goto 104
    p24
124: return
```

Path-1: OUT[S] p0-p1-p2-p3-p24
Path-2: p0-p1-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2
Path-3: p0-p1-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2
Path-4: p0-p1-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2-p3-p24

Path Examples: Basic Block

```
p0: // Block B1
  0: n = 5
  1: i = 0
p1: // goto B2

p2: // Block B2
  0: if i < n goto B4
p3: // goto B7

p12: // Block B7
  0: return
p13:

p4: // Block B4
  0: t4 = 4 * i
  1: t5 = a[t4]
  2: t6 = 4 * i
  3: t7 = b[t6]
  4: if t5 >= t7 goto B6
p5: // goto B5

p6: // Block B5
  0: t8 = 4 * i
  1: t9 = c + t8
  2: t10 = 4 * i
  3: t11 = a[t10]
  4: t12 = 4 * i
  5: t13 = b[t12]
  6: t14 = t11 * t13
  7: *t9 = t14
p7: // goto B3

p8: // Block B6
  0: t15 = 4 * i
  1: t16 = c + t15
  2: *t16 = 0
p9: // goto B3

p10: // Block B3
  0: i = i + 1
p11: // goto B2
```

Path: p0-p1-p2-p4-p5-p8-p9-p10-p11-p2-p3-p12-p13

Uses of Data-flow Analysis

- **Program Debugging**
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*
  - Can a variable may potentially be used without being initialized?
- **Program Optimization**
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.

Data-Flow Analysis Schema

- 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 in consideration
  - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of definitions in the program
  - A particular data-flow value is a set of definitions
- $IN[s]$ and $OUT[s]$: data-flow values *before* and *after* each statement s
- The *data-flow problem* is to find a solution to a set of constraints on $IN[s]$ and $OUT[s]$, for all statements s - this goes first for every block and then for the entire program
- $IN[B]$ and $OUT[B]$:
  - Clearly, $IN[s'] = OUT[s]$ where s' follows s in a block B (due to fall-through within B)
  - So if $IN[B] = IN[s_0]$ and $OUT[B] = OUT[s_n]$ where s_0 and s_n are the *first* and *last* statements of block B, we have $IN[B]$ and $OUT[B]$ as data-flow values *before* and *after* for block B
  - All DFA computations then can be done at the block level

Data-Flow Analysis Schema

- 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 (after the change)
- These safe values may be either *subsets* or *supersets* of actual values, based on the application

Reaching Definitions

Reaching Definitions

- 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
- *Unambiguous* and *ambiguous* definitions of a variable

  a := b+c

  (*unambiguous* definition of *a*)

  ...

  *p := d

  (*ambiguous* definition of *a*, if *p* may point to variables other than *a* as well; hence does not kill the above definition of *a*)

  ...

  a := k-m

  (*unambiguous* definition of *a*; kills the above definition of *a*)

Reaching Definitions

- We compute super-sets 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;
```

Reaching Definitions: How to use them?

- Build *use / def Chains*
- *Constant Propagation*: For a use like

```
n: x = ... v ...
```

  if all definitions that reach n are of the form

```
d: v = c // c is a constant
```

  we can replace v in n by c
- *Un-initialized Variables*: How to detect?
- *Loop-invariant Code Motion*: For

```
d1: a = . . .;
d2: b = . . .;
for (. . .) {
    . . .
    n: x = a + b;
    . . .
}
```

  if all definitions of variables on RHS of n and that reach n are outside the loop like $d1$ and $d2$, n can also be moved outside the loop

Reaching Definitions Problem: DFA Formulation

- 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 $\cup$
  - 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

Reaching Definitions Problem: DFA Formulation

- $GEN[B]$ = set of all definitions inside B that are *visible* immediately after the block - *downwards exposed* definitions
  - If a variable x has two or more definitions 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

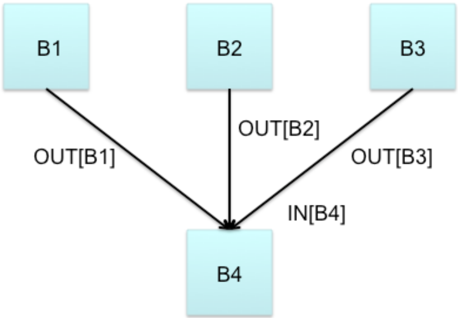
d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

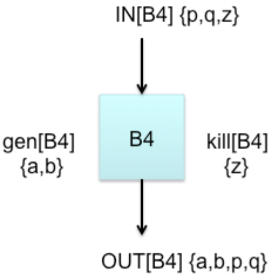
B

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]

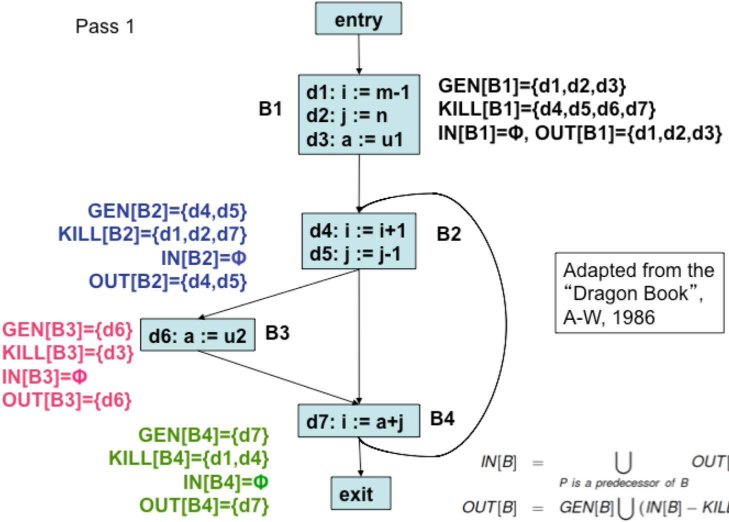
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

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

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

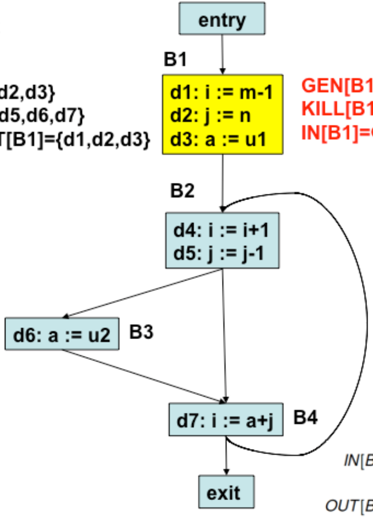
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

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

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}



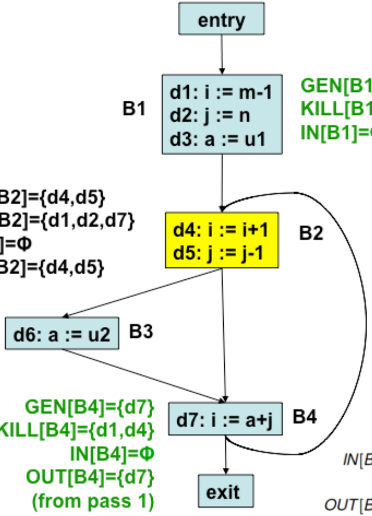
GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

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

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

Reaching Definitions Analysis: An Example - Pass 2.2

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]=Φ
OUT[B2]={d4,d5}

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

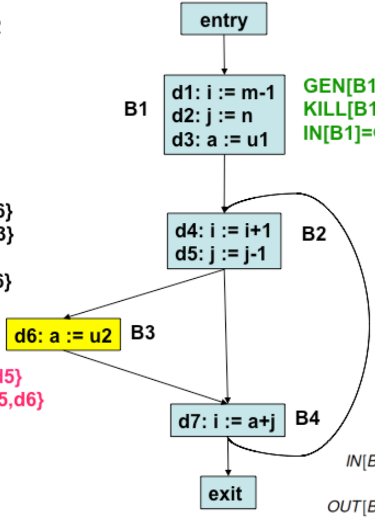
GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]=Φ
OUT[B4]={d7}
(from pass 1)

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

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

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}

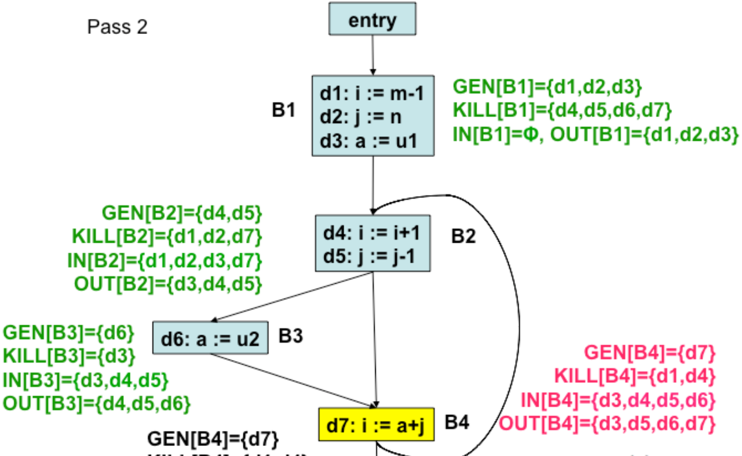
GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]=Φ
OUT[B3]={d6}

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

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

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

Reaching Definitions Analysis: An Example - Pass 2.4

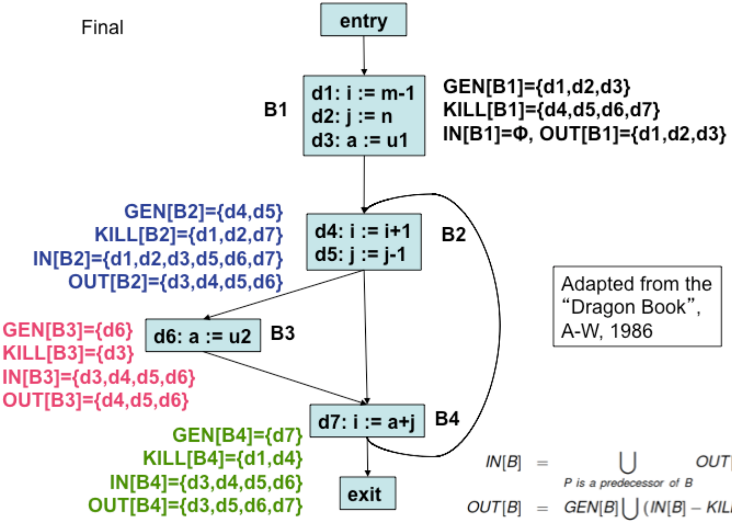


$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

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

Reaching Definitions Analysis: An Example - Final

Final



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 Definition

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

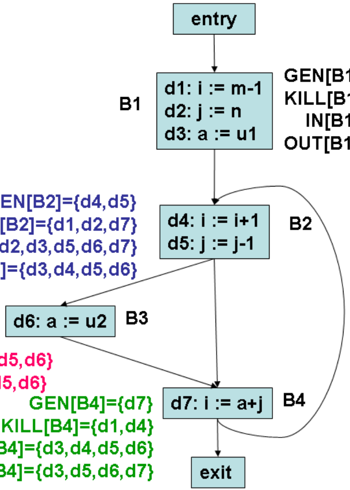
$$\begin{aligned} IN[B] &= \bigcup_{P \ a \ predecessor \ of \ B} OUT[P]; \\ oldout &= OUT[B]; \\ OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]); \end{aligned}$$

    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



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}

	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

DFA: Available Expressions

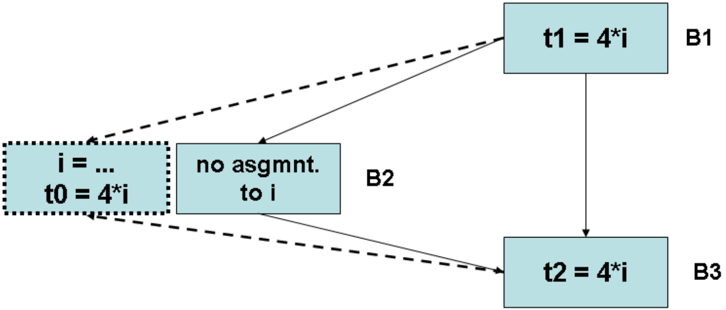
Available Expressions

Available Expression Computation

- *Sets of expressions* constitute the domain of data-flow values
- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not *necessarily cycle-free*) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it *assigns* (or *may assign*) to x or y and does not subsequently recompute $x + y$
- A block *generates* $x + y$, if it definitely evaluates $x + y$, and does not subsequently redefine x or y

Available Expression Computation

- Useful for global common sub-expression elimination
- $4 * i$ is a CSE in $B3$, if it is available at the entry point of $B3$ *that is*, if i is not assigned a new value in $B2$ or $4 * i$ is recomputed after i is assigned a new value in $B2$ (as shown in the dotted box)



Computing e_gen and e_kill

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A   q ▪

x = y + z

p ▪

Computing e_gen[p]

1. A = A **U** {y+z}
2. A = A – {all expressions involving x}
3. e_gen[p] = A

e_kill[q] = A   q ▪

x = y + z

p ▪

Computing e_kill[p]

1. A = A – {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

Available Expression Computation - EGEN and EKILL

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

B:

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

Set of all expressions = {f+1,a+7,b+d,d+c,a+4,e+c,a+b,c+f,e+a}

EGEN[B] = {f+1,b+d,d+c}
EKILL[B] = {a+4,a+b,e+a,e+c,c+f,a+7}

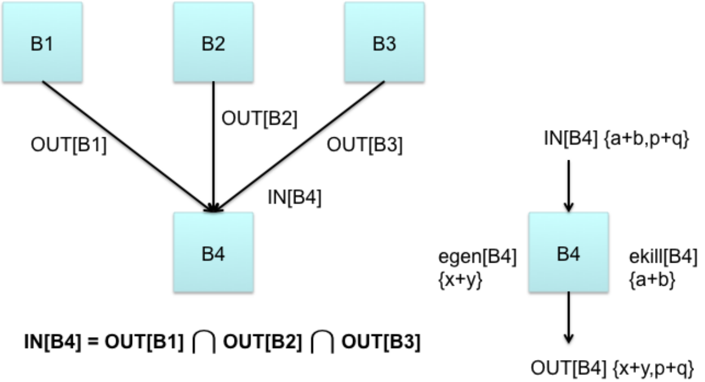
Available Expression Computation - DF Equations

- The data-flow equations

$$
\begin{aligned}
IN[B] &= \bigcap_{P \text{ is a predecessor of } B} OUT[P], \quad B \textit{ not initial} \\
OUT[B] &= e\_GEN[B] \bigcup (IN[B] - e\_KILL[B]) \\
IN[B1] &= \phi \\
IN[B] &= U, \quad \textit{for all } B \neq B1 \ (\textit{initialization only})
\end{aligned}
$$

- $B1$ is the initial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restrictive

Available Expression Computation - DF Equations



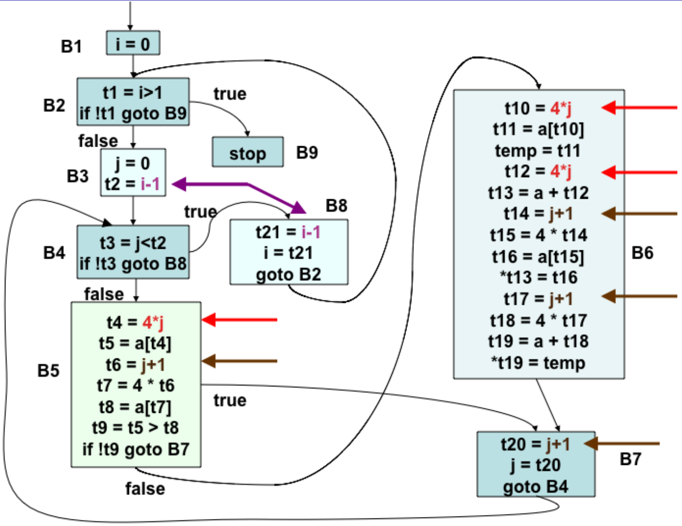
IN[B4] = OUT[B1] ⋂ OUT[B2] ⋂ OUT[B3]

OUT[B4] = egen[B4] ⋃ (IN[B4] – ekill[B4])

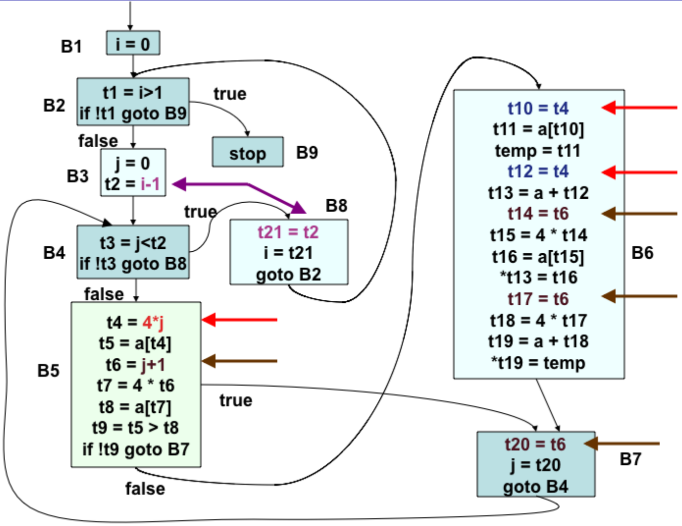
$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P], \; B \text{ not initial}$$

$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$

Available Expression Computation - An Example



Available Expression Computation - An Example



An Iterative Algorithm for Computing Available Expressions

for each block $B \neq B1$ do {
    $OUT[B] = U - e\_KILL[B]$;
}
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
$change = true$;
while $change$ do { $change = false$;
  for each block $B \neq B1$ do {

$$\begin{aligned} IN[B] &= \bigcap_{P \text{ a predecessor of } B} OUT[P]; \\ oldout &= OUT[B]; \\ OUT[B] &= e\_GEN[B] \bigcup (IN[B] - e\_KILL[B]); \end{aligned}$$

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

Live Variables

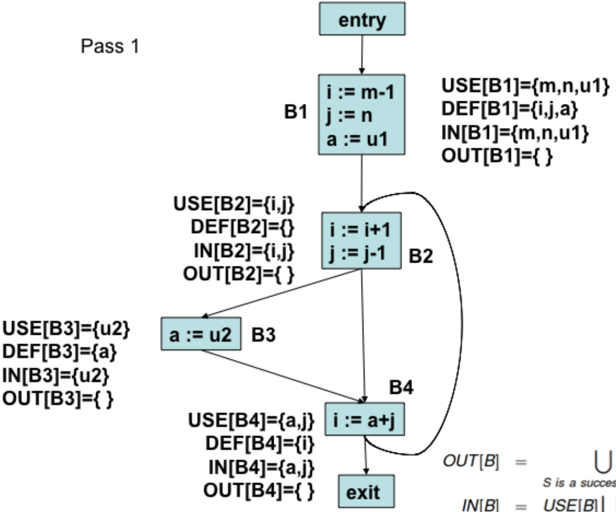
Live Variable Analysis

- 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 em *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

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

Live Variable Analysis: An Example - Pass 1

Pass 1



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

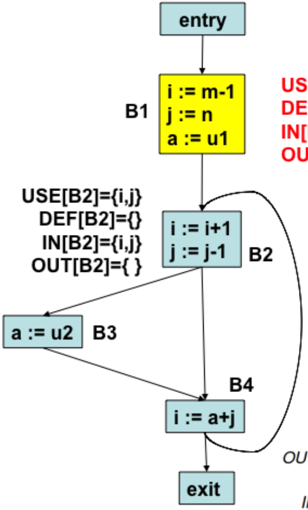
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Module 09

Das & Mitra

DFA
Points, Paths & Use
DFA Schema
DFA Problems
Reaching Definitions
DFA Equations
Example
Algorithm
Bit-Vector
Available Expressions
DFA Equations
Example
Algorithm
Live Variable
DFA Equations
Example
DU Chains
DFA Equations
Copy Propagation
DFA Equations
Algorithm
Example
Module Summary

Pass 2



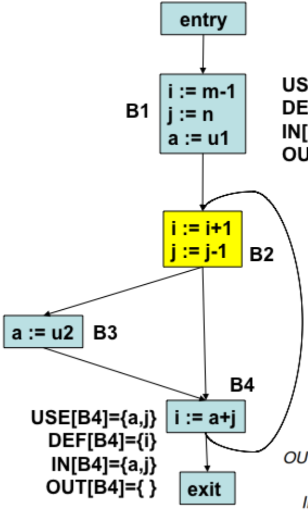
USE[B1]={m,n,u1}
DEF[B1]={i,j,a}
IN[B1]={m,n,u1}
OUT[B1]={i,j}

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

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

Live Variable Analysis: An Example - Pass 2.2

Pass 2



USE[B1]={m,n,u1}
DEF[B1]={i,j,a}
IN[B1]={m,n,u1}
OUT[B1]={i,j}

USE[B2]={i,j}
DEF[B2]={}
IN[B2]={i,j,u2,a}
OUT[B2]={a,j,u2}

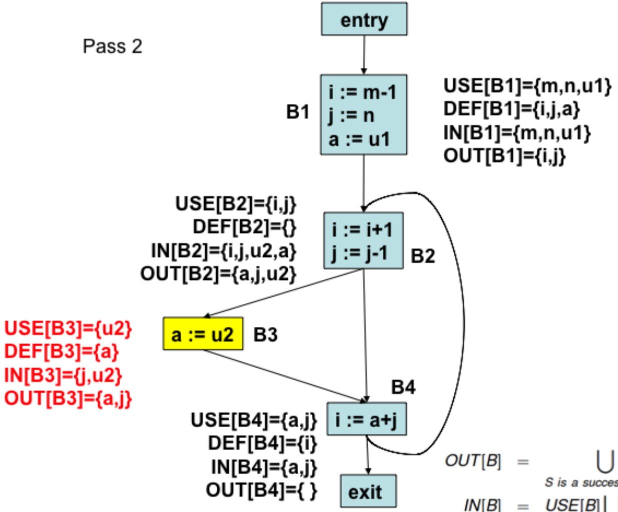
USE[B3]={u2}
DEF[B3]={a}
IN[B3]={u2}
OUT[B3]={ }

USE[B4]={a,j}
DEF[B4]={i}
IN[B4]={a,j}
OUT[B4]={ }

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

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

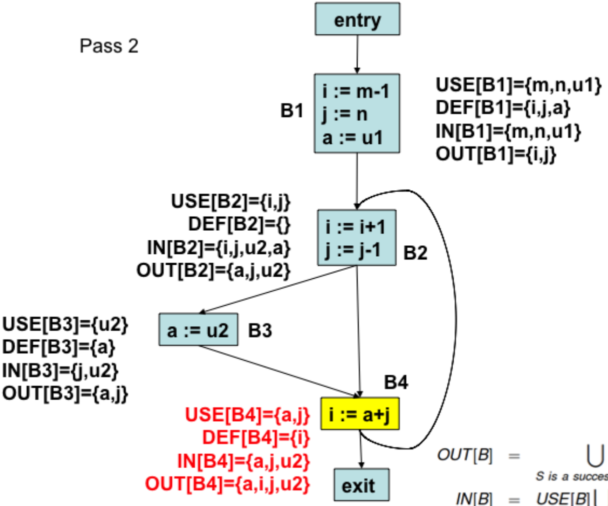
Live Variable Analysis: An Example - Pass 2.3



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

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

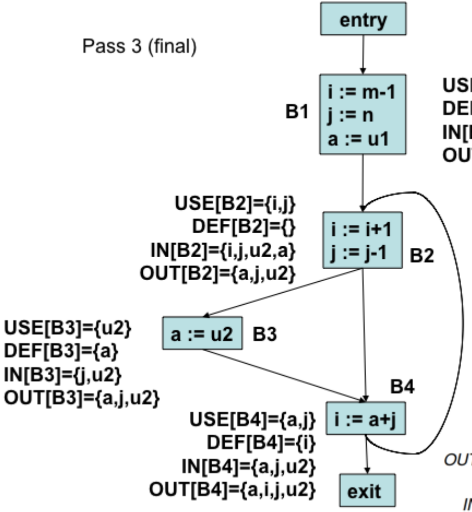
Live Variable Analysis: An Example - Pass 2.4



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

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

Live Variable Analysis: An Example - Final pass



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

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

Definition-Use Chains

DFA: Definition-Use Chains

- For each definition, we wish to attach the statement numbers of the uses of that definition
- Such information is very useful in implementing register allocation, loop invariant code motion, etc.
- This problem can be transformed to the data-flow analysis problem of computing for a point p, the set of uses of a variable (say x), such that there is a path from p to the use of x, that does not redefine x
- This information is represented as sets of $(x; s)$ pairs, where x is the variable used in statement s
- In *live variable* analysis, we need information on whether a variable is used later, but in $(x; s)$ computation, we also need the statement numbers of the uses
- The data-flow equations are similar to that of Live Variable analysis
- Once IN[B] and OUT[B] are computed, d-u chains can be computed using a method similar to that of u-d chains

Data Flow Analysis for (x, s) Pairs

- Sets of pairs (x, s) constitute the domain of data-flow values
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of pairs (x, s), such that statement s uses variable x and the value of x at $IN[B]$ has not been modified along the path from $IN[B]$ to s
- $OUT[B]$ is the set of pairs (x, s), such that statement s uses variable x and the value of x at $OUT[B]$ has not been modified along the path from $OUT[B]$ to s
- $DEF[B]$ is the set of pairs (x, s), such that s is a statement which uses x, s is not in B, and B contains a definition of x
- $USE[B]$ is the set of pairs (x, s), such that s is a statement in B which uses variable x and such that no prior definition of x occurs in B

Data Flow Analysis for (x, s) Pairs

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

Copy Propagation

Copy Propagation

- Eliminate copy statements of the form $s: \ x := y$, by substituting y for x in all uses of x reached by this copy
- Conditions to be checked
  [1] u-d chain of use u of x must consist of s only. Then, s is the only definition of x reaching u
  [2] On every path from s to u, including paths that go through u several times (but do not go through s a second time), there are no assignments to y. This ensures that the copy is valid
- The second condition above is checked by using information obtained by a new data-flow analysis problem
  - $c\_GEN[B]$ is the set of all copy statements, $s : x := y$ in B, such that there are no subsequent assignments to either x or y within B, after s
  - $c\_KILL[B]$ is the set of all copy statements, $s : x := y$, s not in B, such that either x or y is assigned a value in B
  - Let U be the universal set of all copy statements in the program

Copy Propagation - The Data-flow Equations

- $c\_IN[B]$ is the set of all copy statements, $x := y$ reaching the beginning of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path
- $c\_OUT[B]$ is the set of all copy statements, $x := y$ reaching the end of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path

$$
\begin{aligned}
c\_IN[B] &= \bigcap_{P \text{ is a predecessor of } B} c\_OUT[P], \quad B \text{ not initial} \\
c\_OUT[B] &= c\_GEN[B] \bigcup (c\_IN[B] - c\_KILL[B]) \\
c\_in[B1] &= \phi, \quad \text{where } B1 \text{ is the initial block} \\
c\_OUT[B] &= U - c\_KILL[B], \quad \text{for all } B \neq B1 \text{ (initialization only)}
\end{aligned}
$$

Algorithm for Copy Propagation

For each copy, $s : x := y$, do the following

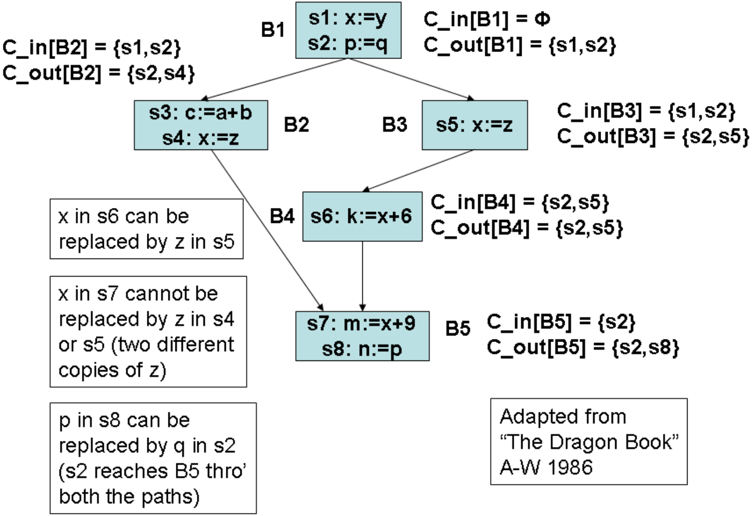
[1] Using the $du - chain$, determine those uses of x that are reached by s

[2] For each use u of x found in (1) above, check that

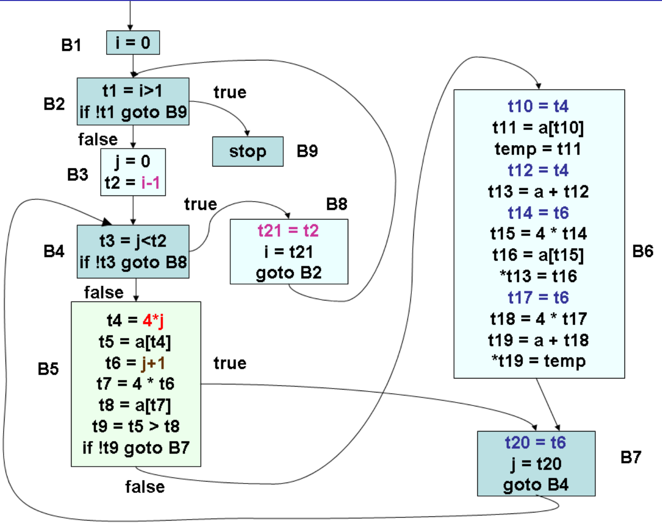
- (i) u-d chain of u consists of s only
- (ii) s is in $c\_IN[B]$, where B is the block to which u belongs. This ensures that
  - s is the only definition of x that reaches this block
  - No definitions of x or y appear on this path from s to B
- (iii) no definitions x or y occur within B prior to u found in (1) above

[3] If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

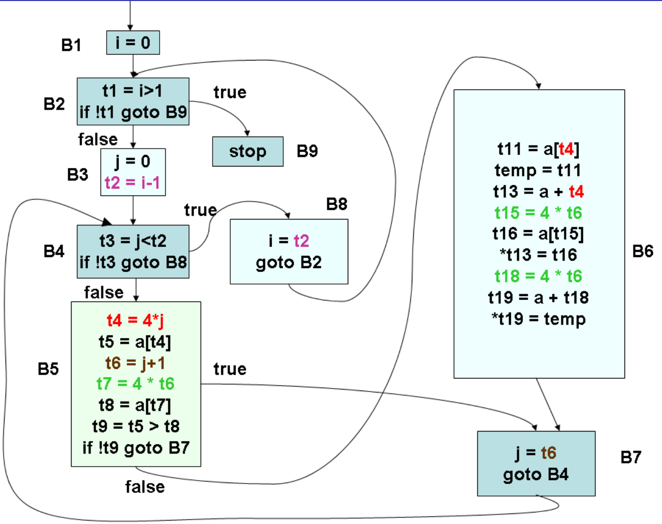
Copy Propagation Example 1



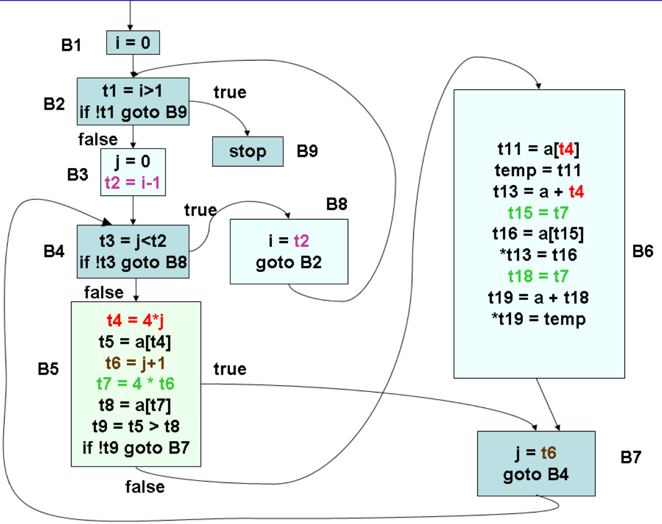
Copy Propagation on Running Example 1.1



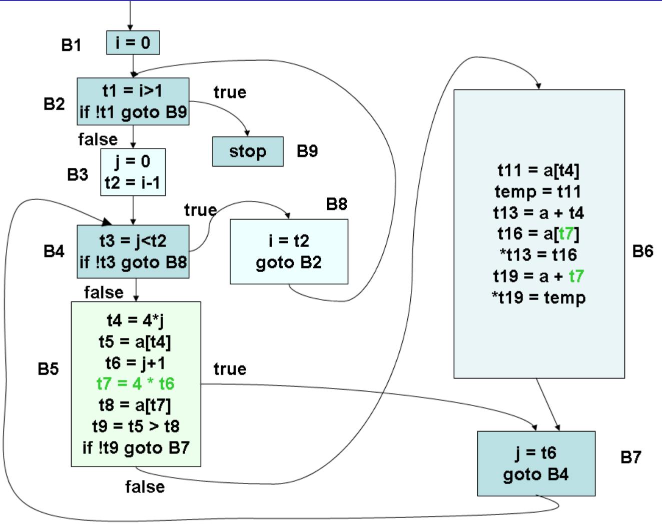
Copy Propagation on Running Example 1.2



GCSE and Copy Propagation on Running Example 1.1



GCSE and Copy Propagation on Running Example 1.2



Module Summary

- Understood Data Flow Analysis (DFA) to estimate various data propagation entities in programs statically
- Understood DFA formulation with forward / backward flow and inclusive / exclusive confluence
- Understood formulation for various DFA solutions for Reaching Definitions, Available Expressions, Live Variables, Def-Use Chains, Copy Propagation etc.
- Understood use of DFA in global optimization