Data-flow Analysis - Part 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

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 $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n - 1$, 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

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

- 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
- $IN[s]$ 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

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 (normally 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

The Reaching Definitions Problem

- 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')

The Reaching Definitions Problem(2)

- Sets of definitions constitute the domain of data-flow values
- 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)

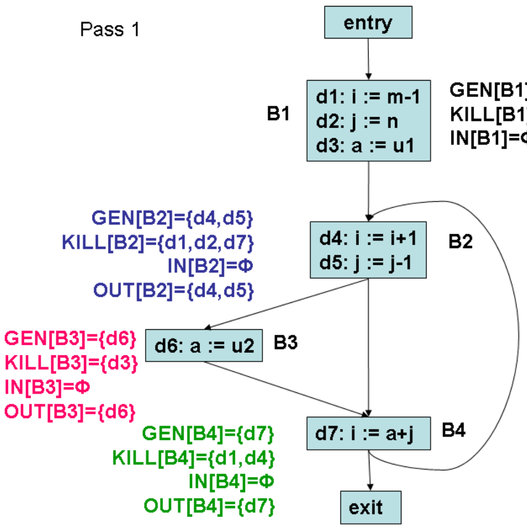
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

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

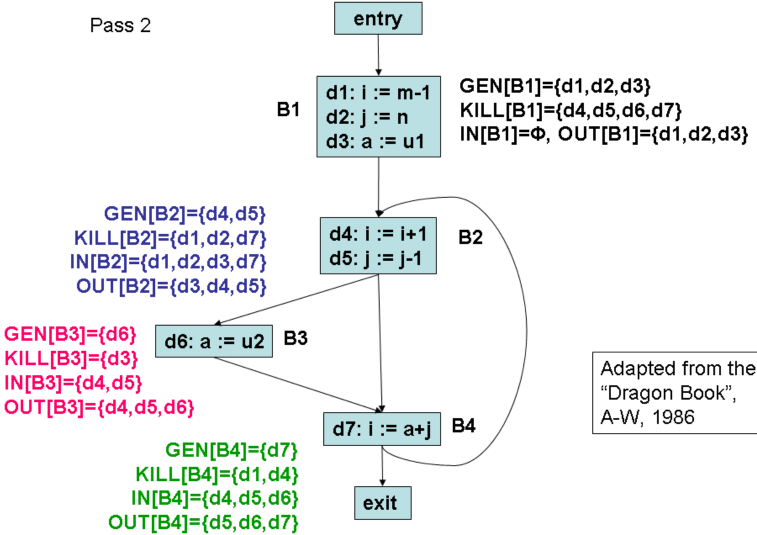
$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- 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 $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B

Reaching Definitions Analysis: An Example - Pass 1

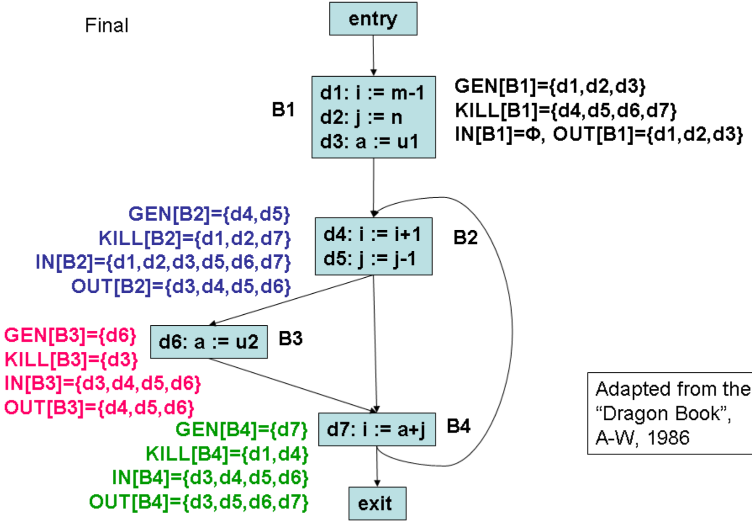


Adapted from the "Dragon Book", A-W, 1986



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

Reaching Definitions Analysis: An Example - Final



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

An Iterative Algorithm for Computing Reaching Definitions

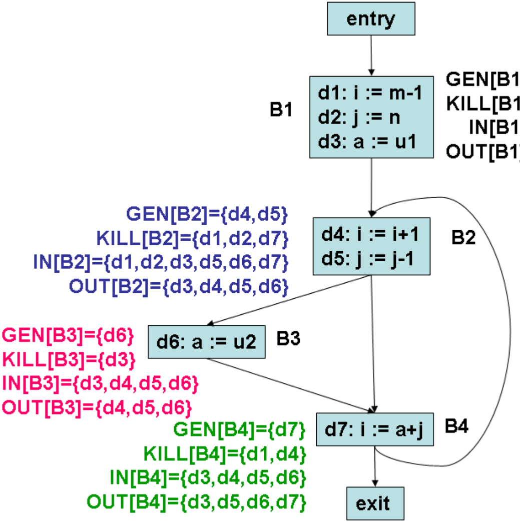
for each block B do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
$change = true$;
while $change$ do { $change = false$;
for each block B do {

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

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

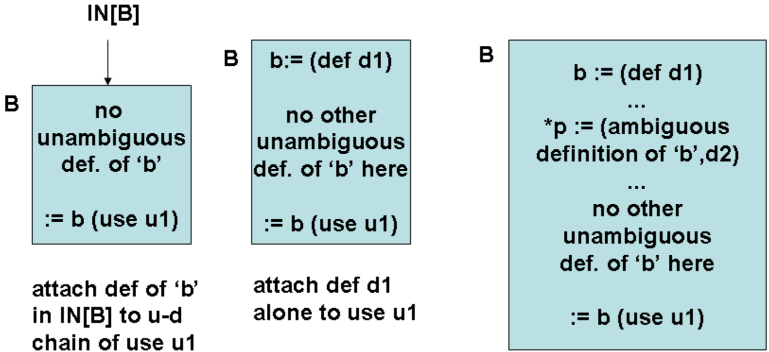
	d1	d2	d3	d4	d5	d6	d7
GEN[B1]=	1	1	1	0	0	0	0
KILL[B1]=	0	0	0	1	1	1	1
IN[B1]=	0	0	0	0	0	0	0
OUT[B1]=	1	1	1	0	0	0	0

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

Use-Definition Chains (u-d chains)

- Reaching definitions may be stored as u-d chains for convenience
- A u-d chain is a list of a use of a variable and all the definitions that reach that use
- u-d chains may be constructed once reaching definitions are computed
- **case 1**: If use $u1$ of a variable b in block B is preceded by no unambiguous definition of b, then attach all definitions of b in $IN[B]$ to the u-d chain of that use $u1$ of b
- **case 2**: If any unambiguous definition of b preceeds a use of b, then *only that definition* is on the u-d chain of that use of b
- **case 3**: If any ambiguous definitions of b precede a use of b, then each such definition for which no unambiguous definition of b lies between it and the use of b, are on the u-d chain for this use of b

Use-Definition Chain Construction



attach def of ‘b’ in IN[B] to u-d chain of use u1

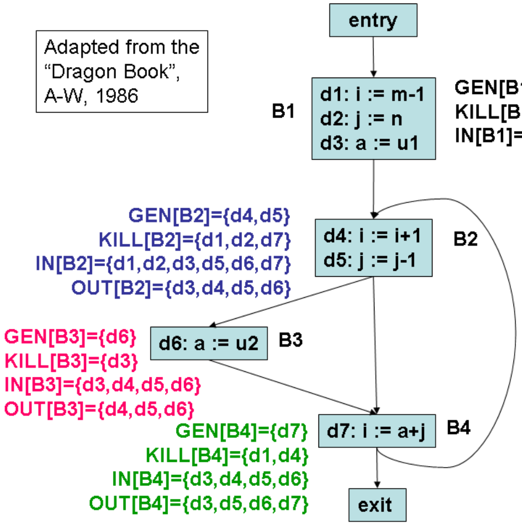
attach def d1 alone to use u1

attach both d1 and d2 to use u1

Three cases while constructing u-d chains from the reaching definitions

Use-Definition Chain Example

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



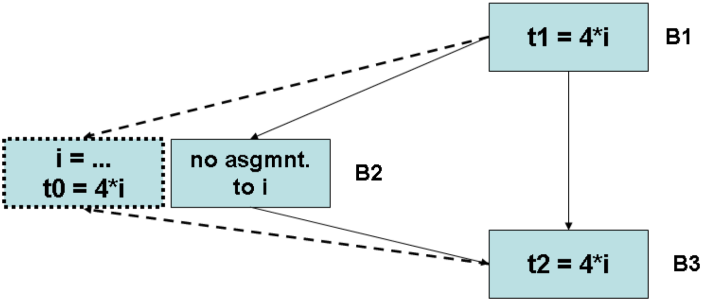
use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	(d3,d6)
(j,d7)	(d5)

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(2)

- Useful for global common sub-expression elimination
- $4 * i$ is a CSE in $B3$, if it is available at the entry point of $B3$ *i.e.,* 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)



Available Expression Computation (3)

- The data-flow equations

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

- $B1$ is the intial 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

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

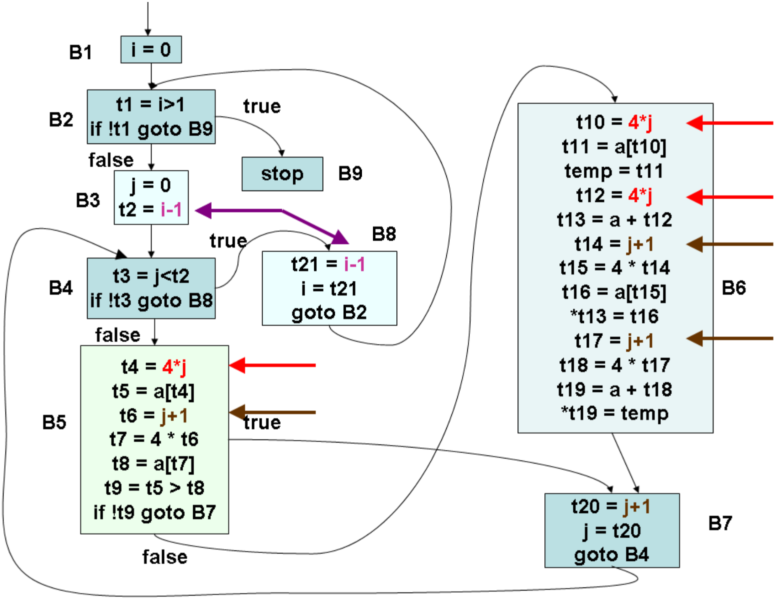
e_kill[q] = A

q ▪
x = y + z
p ▪

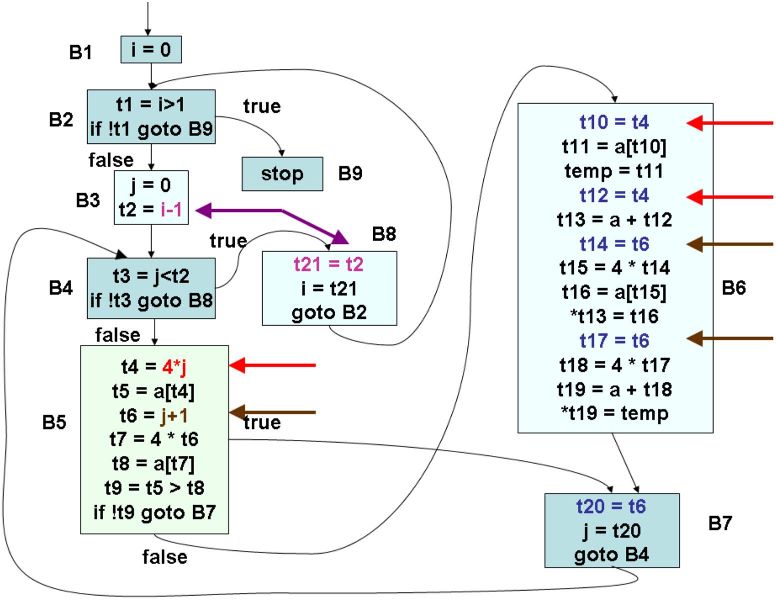
Computing e_kill[p]

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

Available Expression Computation - An Example



Available Expression Computation - An Example (2)



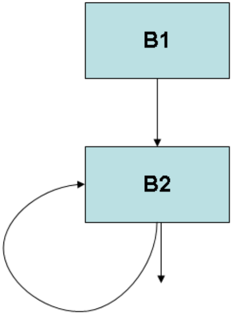
An Iterative Algorithm for Computing Available Expressions

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 {

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

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

Initializing IN[B] to ϕ for all B can be restrictive



Let e_gen[B2] be G and e_kill[B2] be K

IN[B2] = OUT[B1] ∩ OUT[B2]

OUT[B2] = G U IN[B2] – K)

IN^0[B2]=Φ, OUT^1[B2]=G

IN^1[B2]=OUT[B1] ∩ G

OUT^2[B2]=G U ((OUT[B1] ∩ G) – K)

= G U G = G

Note that (OUT[B1] ∩ G) is always smaller than G

IN^0[B2]= ***U***, OUT^1[B2]= ***U*** - K

IN^1[B2]=OUT[B1] ∩ (***U*** – K)

= OUT[B1] - K

OUT^2[B2]=G U ((OUT[B1] - K) – K)

= G U (OUT[B1] - K)

This set OUT[B2] is larger and more intuitive, but still correct

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 *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

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

Live Variable Analysis: An Example