Data-flow Analysis - Part 1

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;
  ```

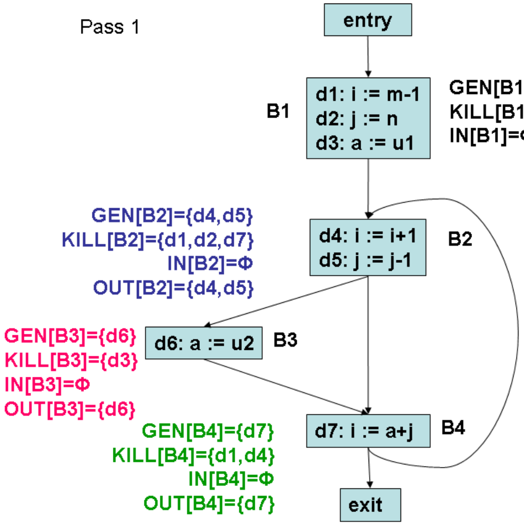
The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

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

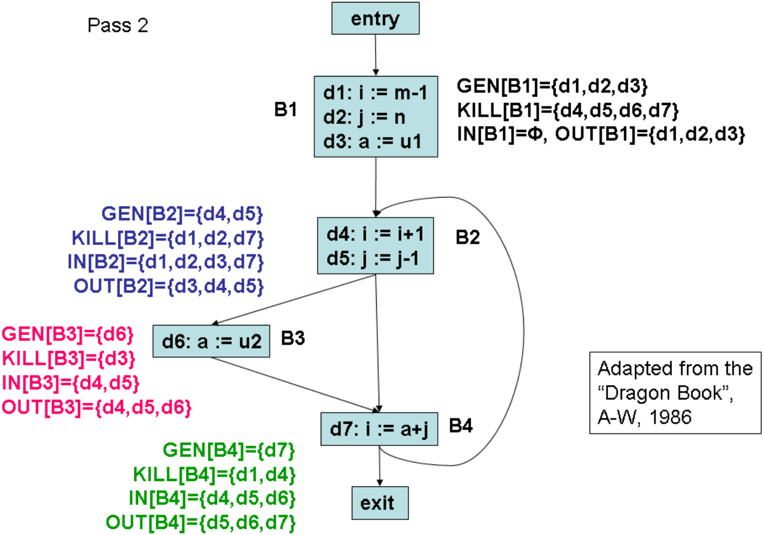
- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B

Pass 1

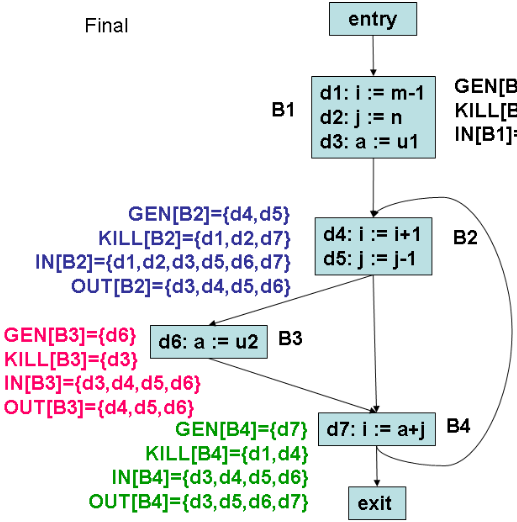


Adapted from the "Dragon Book", A-W, 1986

Reaching Definitions Analysis: An Example - Pass 2



Reaching Definitions Analysis: An Example - Final



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

An Iterative Algorithm for Computing Reaching Definitions

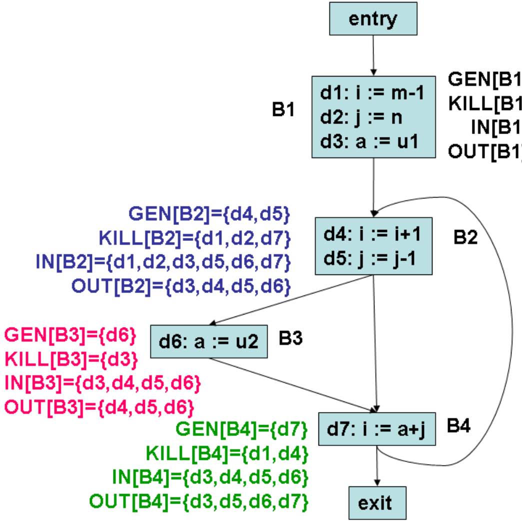
for each block B do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
$change = true$;
while $change$ do { $change = false$;
for each block B do {

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

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