Introduction to
Machine-Independent Optimizations - 2
Data-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations
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 $s$. 
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
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 ’$a$’)

Y.N. Srikant | Data-Flow Analysis
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)

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

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

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

- 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 \(\bigcup\)
  - 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 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:

\[
\begin{align*}
\text{d5: } b &= a + 4 \\
\text{d6: } f &= e + c \\
\text{d7: } e &= b + d \\
\text{d8: } d &= a + b \\
\text{d9: } a &= c + f \\
\text{d10: } c &= e + a
\end{align*}
\]

\[
\begin{align*}
\text{d1: } a &= f + 1 \\
\text{d2: } b &= a + 7 \\
\text{d3: } c &= b + d \\
\text{d4: } a &= d + c
\end{align*}
\]

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

\[ \text{IN}[^{B4}] = \text{OUT}[^{B1}] \cup \text{OUT}[^{B2}] \cup \text{OUT}[^{B3}] \]

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

\[ \text{OUT}[^{B}] = \text{GEN}[^{B}] \cup (\text{IN}[^{B}] - \text{KILL}[^{B}]) \]

\[ \text{OUT}[^{B4}] = \text{gen}[^{B4}] \cup (\text{IN}[^{B4}] - \text{kill}[^{B4}]) \]
Reaching Definitions Analysis: An Example - Pass 1

Pass 1

entry

d1: i := m-1

B1

d2: j := n
d3: a := u1

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}

B2

d4: i := i+1
d5: j := j-1

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

B3

d6: a := u2

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

B4

d7: i := a+j

GEN[B4] = {d7}
KILL[B4] = {d1, d4}
IN[B4] = Φ
OUT[B4] = {d7}

exit

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

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

Y.N. Srikant  Data-Flow Analysis
Reaching Definitions Analysis: An Example - Pass 2.1

Pass 2

entry

B1

d1: i := m-1
d2: j := n
d3: a := u1

B2

d4: i := i+1
d5: j := j-1

B3

d6: a := u2

B4

d7: i := a+j

exit

\[ \text{IN}[B] = \bigcup_{P \text{ is a predecessor of } B} \text{OUT}[P] \]
\[ \text{OUT}[B] = \text{GEN}[B] \bigcup (\text{IN}[B] - \text{KILL}[B]) \]
Reaching Definitions Analysis: An Example - Pass 2.2

Pass 2

entry

d1: i := m-1
d2: j := n
d3: a := u1

B1

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

d4: i := i+1
d5: j := j-1

B2

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

B3

d6: a := u2

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

B4

d7: i := a+j

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

exit
Reaching Definitions Analysis: An Example - Pass 2.3

Pass 2

entry

B1

d1: i := m-1
d2: j := n
d3: a := u1

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

B2

d4: i := i+1
d5: j := j-1

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

B3

d6: a := u2

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

B4

d7: i := a+j

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

OUT[B] = GEN[B] \cup (IN[B] - KILL[B])
Reaching Definitions Analysis: An Example - Pass 2.4

- **Pass 2**
  - **entry**
  - **B1**
    - `d1: i := m-1`
    - `d2: j := n`
    - `d3: a := u1`
    - `GEN[B1]={d1,d2,d3}`
    - `KILL[B1]={d4,d5,d6,d7}`
    - `IN[B1]=Φ, OUT[B1]={d1,d2,d3}`

- **d4: i := i+1**
  - `d5: j := j-1`
  - `GEN[B2]={d4,d5}`
  - `KILL[B2]={d1,d2,d7}`
  - `IN[B2]={d1,d2,d3,d7}`
  - `OUT[B2]={d3,d4,d5}`

- **d6: a := u2**
  - `GEN[B3]={d6}`
  - `KILL[B3]={d3}`
  - `IN[B3]={d3,d4,d5}`
  - `OUT[B3]={d4,d5,d6}`

- **d7: i := a+j**
  - `GEN[B4]={d7}`
  - `KILL[B4]={d1,d4}`
  - `IN[B4]={d3,d4,d5,d6}`
  - `OUT[B4]={d3,d5,d6,d7}`

- **exit**

**Equations**:

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

\[
OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])
\]
Reaching Definitions Analysis: An Example - Final

Final

entry

B1

d1: i := m-1
d2: j := n
d3: a := u1

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

B2

d4: i := i+1
d5: j := j-1

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

B3

d6: a := u2

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

B4

d7: i := a+j

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])

Y.N. Srikant
Data-Flow Analysis
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 { 
      
      \[ IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]; \]

      \[ oldout = OUT[B]; \]

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

      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

**Entry**
- **B1**:
  - d1: i := m-1
  - d2: j := n
  - d3: a := u1

**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

**B2**:
- d4: i := i+1
- d5: j := j-1

**GEN[B2]**:
- {d4, d5}

**KILL[B2]**:
- {d1, d2, d7}

**IN[B2]**:
- {d1, d2, d3, d5, d6, d7}

**OUT[B2]**:
- {d3, d4, d5, d6}

**B3**:
- d6: a := u2

**GEN[B3]**:
- {d6}

**KILL[B3]**:
- {d3}

**IN[B3]**:
- {d3, d4, d5, d6}

**OUT[B3]**:
- {d3, d4, d5, d6}

**B4**:
- d7: i := a+j

**GEN[B4]**:
- {d7}

**KILL[B4]**:
- {d1, d4}

**IN[B4]**:
- {d3, d4, d5, d6}

**OUT[B4]**:
- {d3, d5, d6, d7}

**Exit**

**Final dataflow value sets shown in bit vector format**

- d1 d2 d3 d4 d5 d6 d7

**Adapted from the “Dragon Book”, A-W, 1986**
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
- $4 \times i$ 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 $4 \times i$ 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.

\begin{align*}
e_{\text{gen}}[q] &= A \quad q \cdot \quad x = y + z \quad p \cdot \\
e_{\text{kill}}[q] &= A \quad q \cdot \quad x = y + z \quad p \cdot \\
\end{align*}

**Computing e_gen[p]**
1. $A = A \cup \{y+z\}$
2. $A = A - \{\text{all expressions involving } x\}$
3. $e_{\text{gen}}[p] = A$

**Computing e_kill[p]**
1. $A = A - \{y+z\}$
2. $A = A \cup \{\text{all expressions involving } x\}$
3. $e_{\text{kill}}[p] = A$