Data-flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
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.
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 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 (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 ’\( a \’\))
  - ...  
    - \( \ast p := d \) (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 ’\( a \’\))
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.

```c
if (a==b) a=2; else if (a==b) a=4;
```
The Reaching Definitions Problem (3)

- The data-flow equations (constraints)
  
  \[
  \text{IN}[B] = \bigcup P \text{ is a predecessor of } B \ \text{OUT}[P]
  \]
  
  \[
  \text{OUT}[B] = \text{GEN}[B] \bigcup (\text{IN}[B] - \text{KILL}[B])
  \]
  
  \[
  \text{IN}[B] = \phi, \text{ for all } B \text{ (initialization only)}
  \]

- If some definitions reach \(B_1\) (entry), then \(\text{IN}[B_1]\) is initialized to that set

- Forward flow DFA problem (since \(\text{OUT}[B]\) is expressed in terms of \(\text{IN}[B]\)), confluence operator is \(\cup\)

- \(\text{GEN}[B] = \) set of all definitions inside \(B\) that are “visible” immediately after the block - downwards exposed definitions

- \(\text{KILL}[B] = \) 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

Pass 1

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]=\emptyset, OUT[B1]={d1,d2,d3}

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

d4: i := i+1

d5: j := j-1

B2

B3

d6: a := u2

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

d7: i := a+j

B4

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

exit

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

Y.N. Srikant

Data-flow Analysis
Reaching Definitions Analysis: An Example - Pass 2

Pass 2

entry

B1

\[ \begin{align*}
\text{d1: } & i := m-1 \\
\text{d2: } & j := n \\
\text{d3: } & a := u1
\end{align*} \]

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

B2

\[ \begin{align*}
\text{d4: } & i := i+1 \\
\text{d5: } & j := j-1
\end{align*} \]

B3

\[ \begin{align*}
\text{d6: } & a := u2
\end{align*} \]

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

B4

\[ \begin{align*}
\text{d7: } & i := a+j
\end{align*} \]

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

\text{Adapted from the “Dragon Book”, A-W, 1986}
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

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}

exit

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 \{

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

   $oldout = OUT[B]$;

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

   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

**B1**
- entry
- 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

Final dataflow value sets shown in bit vector format:
- 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]

d1 d2 d3 d4 d5 d6 d7

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 $u_1$ 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 $u_1$ of $b$
- **case 2**: If any unambiguous definition of $b$ precedes 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

IN[B]

B

no unambiguous def. of ‘b’

:= b (use u1)

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

B

b := (def d1)

no other unambiguous def. of ‘b’ here

:= b (use u1)

attach def d1 alone to use u1

B

b := (def d1)

... *p := (ambiguous definition of ‘b’, d2)

... no other unambiguous def. of ‘b’ here

:= b (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

Entry:
- \( d_1 : i := m - 1 \)
- \( d_2 : j := n \)
- \( d_3 : a := u_1 \)

\( \text{GEN[B1]} = \{d_1, d_2, d_3\} \)
\( \text{KILL[B1]} = \{d_4, d_5, d_6, d_7\} \)
\( \text{IN[B1]} = \emptyset, \text{OUT[B1]} = \{d_1, d_2, d_3\} \)

Block B1:
- \( d_4 : i := i + 1 \)
- \( d_5 : j := j - 1 \)

\( \text{GEN[B2]} = \{d_4, d_5\} \)
\( \text{KILL[B2]} = \{d_1, d_2, d_7\} \)
\( \text{IN[B2]} = \{d_1, d_2, d_3, d_5, d_6, d_7\} \)
\( \text{OUT[B2]} = \{d_3, d_4, d_5, d_6\} \)

Block B2:
- \( d_6 : a := u_2 \)

\( \text{GEN[B3]} = \{d_6\} \)
\( \text{KILL[B3]} = \{d_3\} \)
\( \text{IN[B3]} = \{d_3, d_4, d_5, d_6\} \)
\( \text{OUT[B3]} = \{d_4, d_5, d_6\} \)

Block B3:
- \( d_7 : i := a + j \)

\( \text{GEN[B4]} = \{d_7\} \)
\( \text{KILL[B4]} = \{d_1, d_4\} \)
\( \text{IN[B4]} = \{d_3, d_4, d_5, d_6\} \)
\( \text{OUT[B4]} = \{d_3, d_5, d_6, d_7\} \)

Exit:

Use vs. u-d chain:

<table>
<thead>
<tr>
<th>Use</th>
<th>u-d chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (i, d_4) )</td>
<td>( (d_1, d_7) )</td>
</tr>
<tr>
<td>( (j, d_5) )</td>
<td>( (d_2, d_5) )</td>
</tr>
<tr>
<td>( (a, d_7) )</td>
<td>( (d_3, d_6) )</td>
</tr>
<tr>
<td>( (j, d_7) )</td>
<td>( (d_5) )</td>
</tr>
</tbody>
</table>
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 $B_3$, if it is available at the entry point of $B_3$ i.e., if $i$ is not assigned a new value in $B_2$ or $4 \times i$ is recomputed after $i$ is assigned a new value in $B_2$ (as shown in the dotted box)

```

B1

```

```

B2

```

```

B3

```

```

i = ...
t0 = 4*i

no asgmnt. to i

```
Available Expression Computation (3)

- The data-flow equations

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

\[
OUT[B] = e_{\text{gen}}[B] \bigcup (IN[B] - e_{\text{kill}}[B])
\]

\[
IN[B1] = \phi
\]

\[
IN[B] = U, \text{ for all } B \neq B1 \text{ (initialization only)}
\]

- \(B1\) is the intial or entry block and is special because nothing is available when the program begins execution
- \(IN[B1]\) is always \(\phi\)
- \(U\) is the universal set of all expressions
- Initializing \(IN[B]\) to \(\phi\) for all \(B \neq B1\), is restrictive
Computing $e_{\text{gen}}$ and $e_{\text{kill}}$

- 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.

### Computing $e_{\text{gen}}[p]$

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

### Computing $e_{\text{kill}}[p]$

1. $A = A - \{y+z\}$
2. $A = A \cup \{\text{all expressions involving } x\}$
3. $e_{\text{kill}}[p] = A$
Available Expression Computation - An Example (2)

Y.N. Srikant

Data-flow Analysis
An Iterative Algorithm for Computing Available Expressions

for each block $B \neq B1$ do {$OUT[B] = U - e_{\text{kill}}[B]$; }
/* 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 {
    
    $IN[B] = \bigcap_{\text{P a predecessor of } B} OUT[P]$;
    $oldout = OUT[B]$;
    $OUT[B] = e_{\text{gen}}[B] \cup (IN[B] - e_{\text{kill}}[B])$;
    
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}
Initializing $\text{IN}[B]$ to $\phi$ for all $B$ can be restrictive

Let $e_{\text{gen}}[B2]$ be $G$ and $e_{\text{kill}}[B2]$ be $K$

\[
\text{IN}[B2] = \text{OUT}[B1] \cap \text{OUT}[B2]
\]
\[
\text{OUT}[B2] = G \cup (\text{IN}[B2] - K)
\]
\[
\text{IN}^0[B2] = \phi, \quad \text{OUT}^1[B2] = G
\]
\[
\text{IN}^1[B2] = \text{OUT}[B1] \cap G
\]
\[
\text{OUT}^2[B2] = G \cup ((\text{OUT}[B1] \cap G) - K)
\]
\[
= G \cup G = G
\]

Note that $(\text{OUT}[B1] \cap G)$ is always smaller than $G$

----------------------------------------

\[
\text{IN}^0[B2] = u, \quad \text{OUT}^1[B2] = u - K
\]

\[
\text{IN}^1[B2] = \text{OUT}[B1] \cap (u - K)
\]
\[
= \text{OUT}[B1] - K
\]
\[
\text{OUT}^2[B2] = G \cup ((\text{OUT}[B1] - K) - K)
\]
\[
= G \cup (\text{OUT}[B1] - K)
\]

This set $\text{OUT}[B2]$ 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.

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

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

\[
IN[B] = \phi, \text{ for all } B \text{ (initialization only)}
\]
Live Variable Analysis: An Example

Y.N. Srikant

Data-flow Analysis
Definition-Use Chains (d-u 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 LV analysis.
- Once \( \text{IN}[B] \) and \( \text{OUT}[B] \) 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 \(\bigcup\)
- \(\text{USE}[B]\) 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 \(B\)
- \(\text{DEF}[B]\) 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 \(x\)
- \(\text{IN}[B]\) (\(\text{OUT}[B]\), resp.) is the set of pairs \((x, s)\), such that statement \(s\) uses variable \(x\) and the value of \(x\) at \(\text{IN}[B]\) (\(\text{OUT}[B]\), resp.) has not been modified along the path from \(\text{IN}[B]\) (\(\text{OUT}[B]\), resp.) to \(s\)

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

\[
\text{IN}[B] = \text{USE}[B] \bigcup (\text{OUT}[B] - \text{DEF}[B])
\]

\[
\text{IN}[B] = \phi, \text{ for all } B \text{ (initialization only)}
\]
Definition-Use Chain Example

**Data-flow Analysis**

- **B1**
  - \( \text{entry} \)
  - \( s1: i := m-1 \)
  - \( s2: j := n \)
  - \( s3: a := u1 \)
  - \( \text{USE}[B1] = \{(m,s1),(n,s2),(u1,s3)\} \)
  - \( \text{DEF}[B1] = \{(i,s4),(j,s5),(j,s7),(a,s7)\} \)
  - \( \text{IN}[B1] = \{(m,s1),(n,s2),(u1,s3),(u2,s6)\} \)
  - \( \text{OUT}[B1] = \{(i,s4),(j,s5),(u2,s6),(a,s7)\} \)

- **B2**
  - \( s4: i := i+1 \)
  - \( s5: j := j-1 \)
  - \( \text{USE}[B2] = \{(i,s4),(j,s5)\} \)
  - \( \text{DEF}[B2] = \{(j,s7)\} \)
  - \( \text{IN}[B2] = \{(i,s4),(j,s5),(u2,s6),(a,s7)\} \)
  - \( \text{OUT}[B2] = \{(j,s5),(u2,s6),(a,s7),(j,s7)\} \)

- **B3**
  - \( s6: a := u2 \)
  - \( \text{USE}[B3] = \{(u2,s6)\} \)
  - \( \text{DEF}[B3] = \{(a,s7)\} \)
  - \( \text{IN}[B3] = \{(j,s5),(j,s7),(u2,s6)\} \)
  - \( \text{OUT}[B3] = \{(a,s7),(j,s7),(j,s5),(u2,s6)\} \)

- **B4**
  - \( s7: i := a + j \)
  - \( \text{USE}[B4] = \{(a,s7),(j,s7)\} \)
  - \( \text{DEF}[B4] = \{(i,s4)\} \)
  - \( \text{IN}[B4] = \{(a,s7),(j,s7),(u2,s6)\} \)
  - \( \text{OUT}[B4] = \{(a,s7),(i,s4),(j,s5),(u2,s6)\} \)

**Table:**

<table>
<thead>
<tr>
<th>def</th>
<th>d-u chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i,s1)</td>
<td>(s4)</td>
</tr>
<tr>
<td>(j,s2)</td>
<td>(s5)</td>
</tr>
<tr>
<td>(a,s3)</td>
<td>(s7)</td>
</tr>
<tr>
<td>(i,s4)</td>
<td>()</td>
</tr>
<tr>
<td>(j,s5)</td>
<td>(s5,s7)</td>
</tr>
<tr>
<td>(a,s6)</td>
<td>(s7)</td>
</tr>
<tr>
<td>(i,s7)</td>
<td>(s4)</td>
</tr>
</tbody>
</table>
Definition-Use Chain Construction

b := (def d1)
no unambiguous def. of ‘b’

OUT[B]
attach to du-chain of d1, stmts s_i of all use pairs (b,s_i) in OUT[B]

Three cases while constructing d-u chains from the (x,s) pairs

b := (def d1)
other unambiguous def. of ‘b’ here
:= b (use u1)
attach use u1 to du-chain of def d1

b := (def d1)
... *p := (ambiguous definition of ‘b’,d2)
... no other unambiguous def. of ‘b’ here
:= b (use u1)
attach use u1 to du-chains of both def d1 and def d2
An expression $B \text{ op } C$ is very busy or anticipated at a point $p$, if along every path from $p$, we come to a computation of $B \text{ op } C$ before any computation of $B$ or $C$.

- Useful in code hoisting and partial redundancy elimination.
- Code hoisting does not reduce time, but reduces space.
- We must make sure that no use of $B \text{ op } C$ (from X, Y, or Z below) has any definition of $B$ or $C$ reaching it without passing through $p$. 

\[ X = B \text{ op } C \quad Y = B \text{ op } C \quad Z = B \text{ op } C \]

\[ X = T \quad Y = T \quad Z = T \]
Sets of expressions constitute the domain of data-flow values.

Backward flow analysis with $\cap$ as confluence operator.

$V_{\text{USE}}[n]$ is the set of expressions $B \text{ op } C$ computed in $n$ with no prior definition of $B$ or $C$ in $n$.

$V_{\text{DEF}}[n]$ is the set of expressions $B \text{ op } C$ in $U$ (the universal set of expressions) for which either $B$ or $C$ is defined in $n$, prior to any computation of $B \text{ op } C$.

\[
\begin{align*}
\text{OUT}[n] &= \bigcap_{S \text{ is a successor of } n} \text{IN}[S] \\
\text{IN}[n] &= V_{\text{USE}}[n] \cup (\text{OUT}[n] - V_{\text{DEF}}[n]) \\
\text{IN}[n] &= U, \text{ for all } n \text{ (initialization only)}
\end{align*}
\]
Anticipated Expressions - An Example

(a) a+b is anticipated at: entry to 1 and 4  
a+b is not anticipated at: all other points

(b) a+b is anticipated at all points, except at exit of 4 and entry of 5
The Reaching Definitions Problem

- Domain of data-flow values: sets of definitions
- Direction: Forwards
- Confluence operator: $\cup$
- Initialization: $IN[B] = \emptyset$
- Equations:

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

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$
The Available Expressions Problem
- Domain of data-flow values: sets of expressions
- Direction: Forwards
- Confluence operator: $\cap$
- Initialization: $IN[B] = U$
- Equations:

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

$$OUT[B] = e_{gen}[B] \bigcup (IN[B] - e_{kill}[B])$$

$$IN[B1] = \emptyset$$
The Live Variable Analysis Problem

- Domain of data-flow values: sets of variables
- Direction: backwards
- Confluence operator: $\cup$
- Initialization: $IN[B] = \emptyset$
- Equations:

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

\[
IN[B] = USE[B] \cup (OUT[B] - DEF[B])
\]
The Anticipated Expressions (Very Busy Expressions) Problem

- Domain of data-flow values: sets of expressions
- Direction: backwards
- Confluence operator: $\cap$
- Initialization: $IN[B] = U$
- Equations:

\[
\begin{align*}
OUT[B] &= \bigcap_{S \text{ is a successor of } B} IN[S] \\
IN[B] &= V_{\text{USE}}[B] \bigcup (OUT[B] - V_{\text{DEF}}[B])
\end{align*}
\]