Topic 9: Dataflow Analysis

COS 320

Compiling Techniques

Princeton University
Spring 2015

Prof. David August
Analysis and Transformation

- Analysis:
  - Control Flow Analysis
  - Dataflow Analysis

- Transformation:
  - Register Allocation
  - Optimization
    - Machine dependent/independent
    - Local/Global/Interprocedural
    - Acyclic/Cyclic
  - Scheduling
Dataflow Analysis Motivation

Constant Propagation and Dead Code Elimination:

\[ r1 = 4 \]
\[ r2 = r1 + 5 \]
\[ r2 = 9 \]

Needs dominator, liveness, and reaching definition information.
Dataflow Analysis Motivation

Register Allocation:

- Infinite number of registers (virtual registers) must be mapped to a limited number of real registers.
- Pseudo-assembly must be examined by *live variable analysis* to determine which virtual registers contain values which may be used later.
- Virtual registers which are not simultaneously *live* may be mapped onto the same real register.

1. \( r2 = r1 + 1 \)
2. \( r3 = M[r2] \)
3. \( r4 = r3 + 4 \)
4. \( \text{LOAD} \quad r5 = M[r2 + r4] \)
Dataflow Analysis

Three types we will cover:

- Live Variable
  - Live range for register allocation
  - Scheduling
  - Dead code elimination

- Reaching Definitions
  - Constant propagation
  - Constant folding
  - Copy propagation

- Available expressions
  - Common subexpression elimination
Iterative Dataflow Analysis Framework

- These dataflow analyses are all very similar → define a framework.
- Specify:
  - Two set definitions - $A[n]$ and $B[n]$
  - A transfer function - $f(A, B, IN/OUT)$
  - A confluence operator - $\lor$.
  - A direction - FORWARD or REVERSE.
- For forward analyses:
  $$IN[n] = \lor_{p \in PRED[n]} OUT[p]$$
  $$OUT[n] = f(A, B, IN)$$
- For reverse analyses:
  $$OUT[n] = \lor_{s \in SUCC[n]} IN[s]$$
  $$IN[n] = f(A, B, OUT)$$
Definitions

Control Flow Definitions:

- CFG node has *out-edges* leading to *successor nodes*.
- CFG node has *in-edges* coming from *predecessor nodes*.
- For each CFG node \( n \), \( PRED[n] \) = set of all predecessors of \( n \).
- For each CFG node \( n \), \( SUCCE[n] \) = set of all successors of \( n \).
Iterative Dataflow Analysis Framework

- Iterative dataflow analysis equations are applied in an iterative fashion until \( IN \) and \( OUT \) sets do not change.
- Typically done in (FORWARD or REVERSE) topological sort order of CFG for efficiency.
- \( IN \) and \( OUT \) sets initialized to \( \emptyset \).

For each node \( n \) {
  \( IN[n] = OUT[n] = \{} \);
}
Repeat {
  For each node \( n \) in forward/reverse topological order {
    \( IN'[n] = IN[n] \);
    \( OUT'[n] = OUT[n] \);
    \( IN[n], OUT[n] = (Equations) \);
  }
} until \( IN'[n] = IN[n] \) and \( OUT'[n] = OUT[n] \) for all \( n \).
Definitions for Liveness Analysis

Liveness Definitions:

- A source (RHS) register $t$ is a use of $t$.
- A destination (LHS) register $t$ is a definition of $t$.
- A register $t$ is live on edge $e$ if there exists a path from $e$ to a use of $t$ that does not go through a definition of $t$.
- Register $t$ is live-in at CFG node $n$ if $t$ is live on any in-edge of $n$.
- Register $t$ is live-out at CFG node $n$ if $t$ is live on any out-edge of $n$. 
Definitions for Liveness Analysis

Live Variable Analysis Equation:

- Set definition \((A[n])\): \(USE[n]\) - the set of registers that \(n\) uses.
- Set definition \((B[n])\): \(DEF[n]\) - the set of registers that \(n\) defines.
- Transfer function \((f(A, B, OUT))\): \(USE[n] \cup (OUT[n] - DEF[n])\)
- Confluence operator \((\vee)\): \(\cup\)
- Direction: \(REVERSE\)

\[
OUT[n] = \cup_{s \in SUCC[n]} IN[s] \\
IN[n] = USE[n] \cup (OUT[n] - DEF[n])
\]
Live Variable Analysis Example

1:  \( r1 = 0 \)
2:  \( r2 = r1 + 1 \)
3:  \( r3 = r3 + r2 \)
4:  \( r1 = r2 \times 2 \)
5:  \text{branch } r1 < 10, \text{ L1} \)
6:  \text{return } r3

<table>
<thead>
<tr>
<th>Node</th>
<th>( USE )</th>
<th>( DEF )</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Live Variable Application 1: Register Allocation

Register Allocation:
1. Perform live variable analysis.
2. Build *interference graph*.
3. Color interference graph with real registers.
• Node $t$ corresponds to virtual register $t$.

• Edge $\langle t_i, t_j \rangle$ exists if registers $t_i, t_j$ have overlapping live ranges.

• For some node $n$, if $DEF[n] = \{a\}$ and $OUT[n] = \{b_1, b_2, ... b_k\}$, then add interference edges: $\langle a, b_1 \rangle, \langle a, b_2 \rangle, \langle a, b_k \rangle$

Interference Graph For Example:

<table>
<thead>
<tr>
<th>Node</th>
<th>$DEF$</th>
<th>OUT</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1</td>
<td>r1,r3</td>
<td>r3</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2,r3</td>
<td>r1,r3</td>
</tr>
<tr>
<td>3</td>
<td>r3</td>
<td>r2,r3</td>
<td>r2,r3</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
<td>r1,r3</td>
<td>r2,r3</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>r1,r3</td>
<td>r1,r3</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

Virtual registers r1 and r2 may be mapped to same real registers.
Live Variable Application 2: Dead Code Elimination

- Given statement \( s \) with a definition and no side-effects:
  \[
  r_1 = r_2 + r_3, \quad r_1 = M[r_2], \quad \text{or} \quad r_1 = r_2
  \]
  If \( r_1 \) is not live at the end of \( s \), then the \( s \) is dead
- Dead statements can be deleted.
- Given statement \( s \) without a definition or side-effects:
  \[
  r_1 = \text{call FUN_NAME}, \quad M[r_1] = r_2
  \]
  Even if \( r_1 \) is not live at the end of \( s \), it is not dead.

Example:
\[
\begin{align*}
  r_1 &= r_2 + 1 \\
  r_2 &= r_2 + 2 \\
  r_1 &= r_2 + 3 \\
  M[r_1] &= r_2
\end{align*}
\]
Reaching Definition Analysis

Determines whether definition of register $t$ directly affects use of $t$ at some point in program.

**Reaching Definition Definitions:**

- *unambiguous* - instruction explicitly defines register $t$.
- *ambiguous* - instruction may or may not define register $t$.
  - Global variables in a function call.
  - No ambiguous definitions in tiger since all globals are stored in memory.
- Definition of $d$ (of $t$) *reaches* statement $u$ if a path of CFG edges exists from $d$ to $u$ that does not pass through an unambiguous definition of $t$.
- One unambiguous and many ambiguous definitions of $t$ may reach $u$ on a single path.
Reaching Definition Analysis Equation:

- Set definition \( (A[n]) \): \( GEN[n] \) - the set of definition id’s that \( n \) creates.
- Set definition \( (B[n]) \): \( KILL[n] \) - the set of definition id’s that \( n \) kills.
  - \( defs(t) \) - set of all definition id’s of register \( t \).
- Transfer function \( (f(A, B, IN)) \): \( GEN[n] \cup (IN[n] - KILL[n]) \)
- Confluence operator \( (\cup) \): \( \cup \)
- Direction: FORWARD

\[
IN[n] = \cup_{p \in PRED[n]} OUT[p] \\
OUT[n] = GEN[n] \cup (IN[n] - KILL[n])
\]
Reaching Definition Analysis Example

Node | GEN | KILL | IN | OUT | IN | OUT | IN | OUT
-----|-----|------|----|-----|----|-----|----|-----
 1    |     |      |    |     |    |     |    |     
 2    |     |      |    |     |    |     |    |     
 3    |     |      |    |     |    |     |    |     
 4    |     |      |    |     |    |     |    |     
 5    |     |      |    |     |    |     |    |     
 6    |     |      |    |     |    |     |    |     
 7    |     |      |    |     |    |     |    |     
 8    |     |      |    |     |    |     |    |     
Reaching Definition Application 1: Constant Propagation

- Given Statement \( d: a = c \) where \( a \) is constant
- Given Statement \( u: t = a \ op \ b \)
- If statement \( d \) reach \( u \) and no other definition of \( a \) reaches \( u \), then replace \( u \ b \ c \ op \ b \).

Statements 1 and 6 are dead.
Constant Folding

- Given Statement \( d: \; t = a \; \text{op} \; b \)
- If \( a \) and \( b \) are constant, compute \( c \) as \( a \; \text{op} \; b \), replace \( d \) by \( t = c \)

```
2: r3 = 1
3: branch r3 > 5, 6:
4: r3 = r3 + 1
5: goto 3:
7: r1 = 5 + 10
8: M[r3] = r1
```
If $x \circ p y$ is computed multiple times, common subexpression elimination (CSE) attempts to eliminate some of the duplicate computations.

1: $r1 = M[A]$

2: $r2 = r1 + 10$

3: $r3 = M[A]$

4: $r4 = r3 + 1$

5: $r5 = r4 + r2$

Need to track expression propagation $\rightarrow$ available expression analysis
Definitions

- Expression \( x \ op \ y \) is *available* at CFG node \( n \) if, on every path from CFG entry node to \( n \), \( x \ op \ y \) is computed at least once, and neither \( x \) nor \( y \) are defined since last occurrence of \( x \ op \ y \) on path.

- Can compute set of expressions available at each statement using system of dataflow equations.

- Statement \( r_1 = M[r_2] \):
  - *generates* expression \( M[r_2] \).
  - *kills* all expressions containing \( r_1 \).

- Statement \( r_1 = r_2 + r_3 \):
  - *generates* expression \( r_2 + r_3 \).
  - *kills* all expressions containing \( r_1 \).
Iterative Dataflow Analysis Framework

• Specify:
  – Two set definitions - \( A[n] \) and \( B[n] \)
  – A transfer function - \( f (A, B, IN/OUT) \)
  – A confluence operator - \( \lor \).
  – A direction - FORWARD or REVERSE.

• For forward analyses:
  \[
  IN[n] = \lor_{p \in PRED[n]} OUT[p]
  
  OUT[n] = f (A, B)
  \]

• For reverse analyses:
  \[
  OUT[n] = \lor_{s \in SUCC[n]} IN[s]
  
  IN[n] = f (A, B)
  \]
Available Expression Analysis:

- \( exp(t) \) - set of all expressions containing \( t \).
- Set definition \((A[n])\): \( GEN[n] \) - the set of all expressions generated by \( n \).
- Set definition \((B[n])\): \( KILL[n] \) - the set of all expressions that \( n \) kills - \( exp(n) \).
- Transfer function \((f(A, B, IN/OUT))\): \( GEN[n] \cup (IN[n] - KILL[n]) \)
- Confluence operator \((\vee)\): \( \cap \)
  - Use of \( \cup \), required initialization of \( IN \) and \( OUT \) sets to \( \emptyset \).
  - Use of \( \cap \), requires initialization of \( IN \) and \( OUT \) sets to \( U \) (except for \( IN \) of entry node).

- Direction: FORWARD

\[
IN[n] = \cap_{p \in PRE_{D}[n]} OUT[p]
\]
\[
OUT[n] = GEN[n] \cup (IN[n] - KILL[n])
\]
Example

```
1:  r1 = M[A]
2:  r2 = M[B]
3:  r3 = r1 + r2
4:  r4 = r3 + r1
5:  branch r3 > r2
6:  r1 = r1 + 12
7:  r4 = r1 + r2
8:  r5 = r1 + r2
9:  M[r5] = r4
```

<table>
<thead>
<tr>
<th>Node</th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M[A]</td>
<td>r1+r2, r1+12, r3+r1</td>
<td>-</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>M[B]</td>
<td>r1+r2</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>3</td>
<td>r1+r2</td>
<td>r3+r1</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>4</td>
<td>r3+r1</td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>6</td>
<td>r1+r2</td>
<td>r1+r2, r3+r1, r1+12</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>7</td>
<td>r1+r2</td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>8</td>
<td>r1+r2</td>
<td>M[r5]</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>M[A], M[B], M[r5]</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>Node</th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>378, 6, 4</td>
<td>-</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>378</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>3</td>
<td>378</td>
<td>4</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>6</td>
<td>378</td>
<td>378, 4, 6</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>7</td>
<td>378</td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>8</td>
<td>378</td>
<td>9</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>9</td>
<td>1, 2, 9</td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Example

1: \( r_1 = M[A] \)

2: \( r_2 = M[B] \)

3: \( r_3 = r_1 + r_2 \)

4: \( r_4 = r_3 + r_1 \)

5: \( \text{branch } r_3 > r_2 \)

6: \( r_1 = r_1 + 12 \)

7: \( r_4 = r_1 + r_2 \)

8: \( r_5 = r_1 + r_2 \)

9: \( M[r_5] = r_4 \)

<table>
<thead>
<tr>
<th>Node</th>
<th>( GEN )</th>
<th>( KILL )</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>378, 4, 6</td>
<td>-</td>
<td>( U )</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>378</td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>3</td>
<td>378</td>
<td>4</td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>378, 4, 6</td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>7</td>
<td>378</td>
<td></td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>8</td>
<td>378</td>
<td>9</td>
<td>( U )</td>
<td>( U )</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1, 2, 9</td>
<td>( U )</td>
<td>( U )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>IN</th>
<th>OUT</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Given statement \( s: t = x \ \operatorname{op} \ y \):

If expression \( x \ \operatorname{op} \ y \) is available at beginning of node \( s \) then:

1. starting from node \( s \), traverse CFG edges backwards to find last occurrence of \( x \ \operatorname{op} \ y \) on each path from entry node to \( s \).

2. create new temporary \( w \).

3. for each statement \( s': v = x \ \operatorname{op} \ y \) found in (1), replace \( s' \) by:

\[
\begin{align*}
    w &= x \ \operatorname{op} \ y \\
    v &= w
\end{align*}
\]

4. replace statement \( s \) by: \( t = w \)
CSE Example

1: \( r_1 = M[A] \)
2: \( r_2 = M[B] \)
3: \( r_3 = r_1 + r_2 \)
4: \( r_4 = r_3 + r_1 \)
5: branch \( r_3 > r_2 \)
6: \( r_1 = r_1 + 12 \)
7: \( r_4 = r_1 + r_2 \)
8: \( r_5 = r_1 + r_2 \)
9: \( M[r_5] = r_4 \)

\( r_1 + r_2 \) in node 8 is a common subexpression.
Copy Propagation

- Given statement $d$: $a = z$ ($a$ and $z$ are both register temps) \( \rightarrow d \) is a copy statement.
- Given statement $u$: $t = a \text{ op } b$.
- If $d$ reaches $u$, no other definition of $a$ reaches $u$, and no definition of $z$ exists on any path from $d$ to $u$, then replace $u$ by: $t = z \text{ op } b$. 

```
1: r1 = M[A]  
2: r2 = M[B]  
3: r99 = r1 + r2  
3': r3 = r99  
4: r4 = r3 + r1  
5: branch r3 > r2  
6: r1 = r1 + 12  
7: r99 = r1 + r2  
7': r4 = r99  
8: r5 = r99  
9: M[r5] = r4  
```
Sets

- Sets have been used in all the dataflow and control flow analyses presented.
- There are at least 3 representations which can be used:
  - Bit-Arrays:
    * Each potential member is stored in a bit of some array.
    * Insertion, Member is $O(1)$.
    * Assuming set size of $N$ and word size of $W$ - Union (OR) and Intersection (AND) is $O(N/W)$.
  - Sorted Lists/Trees:
    * Each member is stored in a list element.
    * Insertion, Member, Union, Intersection is $O(size)$. (Insertion, Member is $O(\log_2 size)$ in trees.)
    * Better for sparse sets than bit-arrays.
  - Hybrids: - Trees with bit-arrays
    * Use Tree to hold elements containing bit-arrays.
    * Union, Intersection is $O(size/W)$. Insertion, Member is $O(\log_2 size/W)$. 

Basic Block Level Analysis

- To improve performance of dataflow, process at basic block level.
  - Represent the entire basic block by a single *super-instruction* which has any number of destinations and sources.
  - Run dataflow at basic block level.
  - Expand result to the instruction level.

- Example:

  \[
  \begin{align*}
  p: & \quad r_1 = r_2 + r_3 \quad \rightarrow \quad r_1, r_2 = r_2, r_3 \\
  n: & \quad r_2 = r_1
  \end{align*}
  \]
Basic Block Level Analysis

- Example:

  \[ p: \ r_1 = r_2 + r_3 \quad \rightarrow \quad r_1, r_2 = r_2, r_3 \]
  
  \[ n: \ r_2 = r_1 \]

- For reaching definitions:

  \[ OUT[n] = GEN[n] \cup (IN[n] - KILL[n]) \]

  But \( IN[n] = OUT[p] \):

  \[ OUT[n] = GEN[n] \cup ((GEN[p] \cup (IN[p] - KILL[p])) - KILL[n]) \]

  Which (clearly) yields:

  \[ OUT[n] = GEN[n] \cup (GEN[p] - KILL[n]) \cup (IN[p] - (KILL[p] \cup KILL[n])) \]

  So:

  \[ GEN[pn] = GEN[n] \cup (GEN[p] - KILL[n]) \]

  \[ KILL[pn] = KILL[p] \cup KILL[n] \]

- Can we do this at the loop or general region level?
Reducible Flow Graphs Revisited

**Definition**

- A flow graph is reducible iff each edge exists in exactly one class:
  1. Forward edges (forms an acyclic graph where every node is reachable from start node)
  2. Back edges (head dominates tail)

**Algorithm:**

1. Remove all backedges
2. Check for cycles:
   - Cycles: Irreducible.
   - No Cycles: Reducible.

**Think:**

- All loop entry arcs point to header.
Motivation:

- Structured programs are always reducible programs.
- Reducible programs are not always structured programs.
- Exploit the structured or reducible property in dataflow analysis.

Structures:

- Lists of instructions
- Conditionals/Hammocks
- While Loops (no breaks)

Method:

- Represent structures by a single *super-instruction* which has any number of destinations and sources.
- Run dataflow at structure level.
- Expand result to the instruction level.
Structured Program Analysis

- Lists of instructions - Basic Blocks!

\[ GEN[pn] = GEN[n] \cup (GEN[p] - KILL[n]) \]

\[ KILL[pn] = KILL[p] \cup KILL[n] \]

- Conditionals/Hammocks

\[ GEN[lr] = GEN[l] \cup GEN[r] \]

\[ KILL[lr] = KILL[l] \cap KILL[r] \]

- While Loops

\[ GEN[loop] = GEN[l] \]

\[ KILL[loop] = KILL[l] \]

Try this on an irreducible flow graph...
Conservative Approximations Example

Register Allocation:

0: $r1 = 1$

1: branch ???

2: $r1 = r1 + 1$

3: branch $r1 < 3$

4: branch $r1 < 5$

5: $= r1$

6: $= r2$
New Dataflow Analysis

0: \( r1 = 1 \)

1: \( \text{branch ??} \)

2: \( r1 = r1 + 1 \)

3: \( \text{branch } r1 < 3 \)

4: \( \text{branch } r1 < 5 \)

5: \( = r1 \)

6: \( = r2 \)
Limitation of Dataflow Analysis

1: \( r_1 = r_2 \times r_2 \)

2: \( r_3 = r_1 + r_2 \)

3: \( \text{branch } r_3 \geq r_2 \)

4: \( = r_1 \)

5: \( = r_3 \)