Dataflow Analysis, cont.

Announcements

- HW1 is posted, due January 28th
- You can work individually or in teams of 2
  - Set up teams in Submitty
  - Ask questions on forum!
  - Upload in Submitty
- Change in office hours: Wed Noon-2pm or by appointment

Outline of Today’s Class

- Building CFG from 3-address code
- Local analysis vs. global analysis
- The four classical dataflow analysis problems
  - Reaching definitions
  - Live variables
  - Available expressions
  - Very busy expressions
- Reading:
  - Dragon Book, Chapter 9.2

Building the Control Flow Graph

Step 1: Partition Code Into Basic Blocks

1. Determine the leader statements:
   (i) First program statement
   (ii) Targets of conditional or unconditional goto’s
   (iii) Any statement following a goto
2. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program
**Question. Find the Leader**

Statements

1. `sum = 0`
2. `i = 1`
3. `if i > n goto 15`
4. `t1 = addr(a) - 4`
5. `t2 = i*4`
6. `t3 = t1[t2]`
7. `t4 = addr(a) - 4`
8. `t5 = i*4`
9. `t6 = t5[t5]`
10. `t7 = t3*t6`
11. `t8 = sum + t7`
12. `sum = t8`
13. `i = i + 1`
14. `goto 3`
15. `...`

Spring 19 CSCI 4450/6450, A Milanova

**Step 2. Add Control Flow Edges**

There is a directed edge from basic block $B_1$ to block $B_2$ if $B_2$ can immediately follow $B_1$ in some execution sequence.

Determine edges as follows:

- There is an edge from $B_1$ to $B_2$ if $B_2$ follows $B_1$ in three address code, and $B_1$ does not end in an unconditional `goto`.
- There is an edge from $B_1$ to $B_2$ if there is a `goto` from the last statement in $B_1$ to the first statement in $B_2$.

Spring 19 CSCI 4450/6450, A Milanova

**Local Analysis vs. Global Analysis**

**Local analysis:** analysis within basic block

- Enables optimizations such as local common subexpression elimination, dead code elimination, constant propagation, copy propagation, etc.

**Global analysis:** beyond the basic block

- Enables optimizations such as global common subexpression elimination, dead code elimination, constant propagation, loop optimizations, etc.

Spring 19 CSCI 4450/6450, A Milanova

**Question. Add Control Flow Edges**

Statements

1. `sum = 0`
2. `i = 1`
3. `if i > n goto 15`
4. `t1 = addr(a) - 4`
5. `t2 = i*4`
6. `t3 = t1[t2]`
7. `t4 = addr(a) - 4`
8. `t5 = i*4`
9. `t6 = t5[t5]`
10. `t7 = t3*t6`
11. `t8 = sum + t7`
12. `sum = t8`
13. `i = i + 1`
14. `goto 3`
15. `...`

Spring 19 CSCI 4450/6450, A Milanova

**Local Analysis: Local Common Subexpression Elimination**

1. $a = y+2$ $y+2$ is “available” in $a$ after execution of statement 1
2. $z = x+w$ $y+2$ in $a$, $x+w$ in $z$
3. $x = y+2$ $y+2$ is available in $a$ but $x+w$ is no longer available
4. $z = b+c$ $y+2$, $b+c$
5. $b = y+2$ $y+2$ is available in $a$, but $b+c$ is no longer available

Spring 19 CSCI 4450/6450, A Milanova

**Question. Run Local Common Subexpression Elimination**

1. `t1 = 4 * i`
2. `t2 = a [ t1 ]`
3. `t3 = 4 * i`
4. `t4 = b [ t3 ]`
5. `t5 = t2 * t4`
6. `t6 = prod * t5`
7. `prod = t6`
8. `t7 = i + 1`
9. `i = t7`
10. `if i <= 20 goto 1`

Spring 19 CSCI 4450/6450, A Milanova
Local Analysis: Dead Code Elimination

1. \( a = y + 2 \)  
2. \( z = x + w \)  
3. \( x = a \)  
4. \( z = b + c \)  
5. \( b = a \)

\( z \) is redefined at 4, and was never used on the way from 2 to 4; thus \( 2. z = x + w \) is “dead code”

After Local Common Subexpression and Dead Code Elimination

1. \( a = y + 2 \)  
2. \( z = x + w \)  
3. \( x = y + 2 \)  
4. \( z = b + c \)  
5. \( b = y + 2 \)

Local Constant Propagation

1. \( t_1 = 1 \)  
2. \( a = t_1 \)  
3. \( t_2 = 1 + a \)  
4. \( k = t_2 \)  
5. \( t_3 = cvtoreal(k) \)  
6. \( t_4 = 6.2 + t_3 \)  
7. \( t_3 = t_4 \)

Arrays and Pointers Make Things Harder

- Consider:
  1. \( x = a[k] \);
  2. \( a[j] = y \);
  3. \( z = a[k] \);
- Can we transform this code into:
  1. \( x = a[k] \);
  2. \( a[j] = y \);
  3. \( z = x \);

Local Analysis vs. Global Analysis

- Local analysis is generally easy – a single path from basic block entry to basic block exit
- Global analysis is hard – multiple paths, across basic blocks
  - Control flow splits and merges at if-then-else
  - Loops!

Dataflow Analysis

- Collects information for all inputs, along all execution paths
  - Control splits and control merges
  - Loops (control goes back)
- Dataflow analysis is a powerful framework
- We can define many different kinds of dataflow analysis
1. Control-flow graph (CFG):
   \( G = (N, E, 1) \)
   - Nodes are basic blocks

2. Data

3. Dataflow equations
   \[ \text{out}(j) = (\text{in}(j) - \text{kill}(j)) \cup \text{gen}(j) \]
   (\text{gen} and \text{kill} are parameters)

4. Merge operator \( \mathcal{V} \)
   \[ \text{in}(j) = \mathcal{V} \text{out}(i) \]
   \( i \) is predecessor of \( j \)

---

**Reaching Definitions**

- **Definition** A statement that may change the value of a variable (e.g., \( x = y + z \))
- \((x, k)\) denotes definition of \( x \) at node \( k \)
- A definition \((x, k)\) reaches node \( n \) if there is a path from \( k \) to \( n \), free of a definition of \( x \)

---

**Live Uses of Variables**

- **Use** Appearance of a variable as an operand of a 3-address statement (e.g., \( x \) in \( y = x + 4 \))
- A use of a variable \( x \) at node \( n \) is **live on exit** from \( k \), if there is a path from \( k \) to \( n \) clear of definition of \( x \)

---

**Def-use Enable Optimizations**

- Dead code elimination (Def-use)
- Code motion (Use-def)
- Constant propagation (Use-def)
- Strength reduction (Use-def)
- Test elision (Use-def)
- Copy propagation (Def-use)

Aside: Def-use enables dataflow-based testing. (We mentioned in Principles)
Question. What are the Def-use Chains that start at 2?

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) – 4
5. t2 = i * 4
6. i = i + 1

Answer: (2, 3)
(2, 5)
(2, 6)

Def-use Enables Dead Code Elimination

1. sum = 0
2. i = 1
3. if t2 > t9 goto 15
4. t3 = t1[t2]
5. t7 = t3 * t3
6. sum = sum + t7
7. t2 = t2 + 4

After code motion, strength reduction, test elision and constant propagation, the def-use links from 2.i=1 disappear. Thus, 2.i=1 becomes dead code.

Use-def Enables Constant Propagation

What are the use-def chains that originate at 6?

1. i = 1
2. i = 2
3. i = 3
4. p = i*2
5. i = 1
6. q = 3+3 = 8

Answer: (6, 1)
(6, 5)

“Hot” Applications of Def-use

- Security!
- Encryption inference
  - To encrypt data in MapReduce programs
  - E.g., sums weight, BP over patients to compute average. Use of weight, BP is +, thus AH
- Protocol inference for multi-party computation
  - Different protocols implement operations at different costs

Problem 1. Reaching Definitions (Reach)

- Problem statement: for each CFG node n, compute the set of definitions (x, k) that may reach n
- First, define data (i.e., the dataflow facts) to propagate
  - Primitive dataflow facts are definitions (x, k)
  - Reach propagates sets of definitions, e.g., {(i,1), (p,4)}

Reaching Definitions (Reach)

- Next, define the dataflow equations (i.e., effect of code at node j on incoming dataflow facts)
  \[ j: \quad x = y+z \]
  \[ \text{kill}(j): \text{all definitions of } (x, \_), \quad \text{gen}(j): \text{this definition of } x, (x, j) \]
  \[ \text{out}(j) = (\text{in}(j) \setminus \text{kill}(j)) \cup \text{gen}(j) \]
- E.g., if in(4) = \{(x, 1), (y, 2), (x, 3)\}
  - Node 4 is: \( x = y+z \)
  - Then out(4) = \{(y, 2), (x, 4)\}
Reaching Definitions (Reach)

Next, define the merge operator $V$ (i.e., how to combine data from incoming paths)

For Reach, $V$ is the set union $\cup$

$$in(j) = \{ \cup out(i) \mid i \text{ is predecessor of } j \}$$

E.g., if $out(2) = \{(x, 1), (y, 2)\}$ and $out(3) = \{(x, 3)\}$ and 2 and 3 are predecessors of 4

$in(4) = \{(x, 1), (x, 3), (y, 2)\}$

Reach: Dataflow Equations

1. $x=5$
   $in(1) = \varnothing$
   $out(1) = \{(x,1)\}$

2. $y=1$
   $in(2) = \{(x,1)\}$
   $out(2) = \{(x,1), (y,2)\}$

3. $x>=2$
   $in(3) = \{(x,1), (x,5), (y,2), (y,4)\}$
   $out(3) = \{(x,1), (x,5), (y,2), (y,4)\}$

4. $y=x*y$
   $in(4) = \{(x,1), (x,5), (y,2), (y,4)\}$
   $out(4) = \{(x,1), (x,5), (y,4)\}$

5. $x=x-1$
   $in(5) = \{(x,1), (x,5), (y,4)\}$
   $out(5) = \{(x,5), (y,4)\}$

6. goto 3
   $in(6) = \{(x,5), (y,4)\}$
   $out(6) = \{(x,5), (y,4)\}$

7. ...
   $in(7) = \{(x,1), (x,5), (y,2), (y,4)\}$
   $out(7) = \{(x,1), (x,5), (y,2), (y,4)\}$

Reach: Solution of Equations

1. $x=5$
   $in(1) = \varnothing$
   $out(1) = \{(x,1)\}$

2. $y=1$
   $in(2) = \{(x,1)\}$
   $out(2) = \{(x,1), (y,2)\}$

3. $x>=2$
   $in(3) = \{(x,1), (x,5), (y,2), (y,4)\}$
   $out(3) = \{(x,1), (x,5), (y,2), (y,4)\}$

4. $y=x*y$
   $in(4) = \{(x,1), (x,5), (y,2), (y,4)\}$
   $out(4) = \{(x,1), (x,5), (y,4)\}$

5. $x=x-1$
   $in(5) = \{(x,1), (x,5), (y,4)\}$
   $out(5) = \{(x,5), (y,4)\}$

6. goto 3
   $in(6) = \{(x,5), (y,4)\}$
   $out(6) = \{(x,5), (y,4)\}$

7. ...
   $in(7) = \{(x,1), (x,5), (y,2), (y,4)\}$
   $out(7) = \{(x,1), (x,5), (y,2), (y,4)\}$

Reach: Live Uses of Variables (Live)

We say that a variable $x$ is “live on exit from node $j$” if there is a live use of $x$ on exit from $j$ (recall the definition of “live use of $x$ on exit from $j$”)

Problem statement: for each node $n$, compute the set of variables that may be live on exit from $n$.

1. $x=2$; $y=4$; $z=1$; if $(y>x)$ then 5. $z=y$; else 6. $z=x*y$; 7. $x=z$; What variables are live on exit from statement 3? Statement 1?
**Live Example**

1. \(x = 2\)
2. \(y = 4\)
3. \(x = 1\)
4. \(y > x\)
5. \(z = y\)
6. \(z = y \times y\)
7. \(x = z\)

**Live Uses of Variables (Live)**

- **Data**
  - Primitive facts: variables \(x\)
  - Propagates sets: \(\{x, y, z\}\)
- **Dataflow equations. At** \(j\): \(x = y + z\)
  - \(\text{kill}_{\{x\}}(j): \{x\}\)
  - \(\text{gen}_{\{y, z\}}(j): \{y, z\}\)
- **Merge operator: set union** \(\cup\)

**Available Expressions**

- An expression \(x \ op \ y\) is available at program point \(n\) if **every** path from entry to \(n\) evaluates \(x \ op \ y\), and after every evaluation prior to reaching \(n\), there are NO subsequent assignments to \(x\) or \(y\)

**Problem 3. Available Expressions (Avail)**

- Problem statement: For every node \(n\), compute the set of expressions that are available at \(n\)

**Avail Enables Global Common Subexpressions**

- \(z = a \times b\)
- \(g = a + b\)
- \(s = a \times 2\)
- \(u = a \div 2\)
- \(w = a \times b\)
Avail Enables Global Common Subexpressions

Can we eliminate \( w = a \times b \)?

Available Expressions (Avail)

- Data?
  - Primitive dataflow facts are expressions, e.g., \( x+y \), \( a \times b \), \( a+2 \)
  - Analysis propagates sets of expressions, e.g., \( \{ x+y, a \times b \} \)
- Dataflow equations at \( j \): \( x = y \text{ op } z \)?
  - \( \text{out}_{Av}(i) = (\text{in}_{Av}(i) \setminus \text{kill}_{Av}(i)) \cup \text{gen}_{Av}(i) \)
  - \( \text{kill}_{Av}(i) \): all expressions with operand \( x \): \( (x \text{ op }_), (_, \text{ op } x) \)
  - \( \text{gen}_{Av}(i) \): new expression: \( \{ (y \text{ op } z) \} \)

Available Expressions (Avail)

- Merge operator?
  - For Avail, it is set intersection \( \cap \)

\[ \text{in}_{Av}(j) = \left\{ \text{out}_{Av}(i) \mid i \text{ is predecessor of } j \right\} \]

Example

1. \( y = a + b \)
2. \( x = a \times b \)
3. if \( y \leq a \times b \)
   4. \( a = a + 1 \)
   5. \( x = a \times b \)
   6. goto 3
   7. _

Very Busy Expressions

An expression \( x \text{ op } y \) is very busy at node \( n \), if along EVERY path from \( n \) to the end of the program, we come to a computation of \( x \text{ op } y \) BEFORE any redefinition of \( x \) or \( y \).
Problem 4. Very Busy Expressions (VeryB)

Problem Statement: For each node $n$, compute the set of expressions that are very busy on exit from $n$.

Q: What is the data?
Q: What are the equations?
Q: What is $\text{gen}_i(j)$?
Q: What is $\text{kill}_i(j)$?
Q: What is the merge operator?

Very Busy Expressions (VeryB)

Data?
- Primitive dataflow facts are expressions, e.g., $x+y, a*b$
- Analysis propagates sets of expressions, e.g., $\{x+y, a*b\}$

Dataflow equations at $j$: $x = y \text{ op } z$?
- $\text{in}(j) = \text{gen}(j) \cup (\text{out}(j) – \text{kill}(j))$
- $\text{kill}(j)$: all expressions with operand $x$: $(x \text{ op } _), (_ \text{ op } x)$
- $\text{gen}(j)$: new expression: $(y \text{ op } z)$

Very Busy Expressions

Merge operator?
- For VeryB, it is set intersection $\cap$

$$\text{out}_{vb}(j) = \{ \cap \text{in}_{vb}(i) | i \text{ is successor of } j \}$$

Backward, must dataflow problem

Dataflow Analysis Problems

<table>
<thead>
<tr>
<th>May Analyses</th>
<th>Must Analyses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Analyses</td>
<td>Reaching Definitions</td>
</tr>
<tr>
<td>Backward Analyses</td>
<td>Live Uses of Variables</td>
</tr>
</tbody>
</table>

Similarities

In all cases, analysis operates on a finite set $D$ of primitive dataflow facts:
- Reach: $D$ is the set of all definitions in the program:
  - e.g., $(x,1), (y,2), (x,4), (y,5)$
- Avail and VeryB: $D$ is the set of all arithmetic expressions:
  - e.g., $a+b, a*b, a+1$
- Live: $D$ is the set of all variables
  - e.g., $x, y, z$
- Solution at node $n$ is a subset of $D$ (a definition either reaches node $n$ or it does not reach node $n$)
Dataflow equations (i.e., transfer functions) for forward problems have generic form:

\[ \text{out}(j) = F_j(\text{in}(j)) = (\text{in}(j) - \text{kill}(j)) \cup \text{gen}(j) = (\text{in}(j) \cap \text{pres}(j)) \cup \text{gen}(j) \]

\[ \text{in}(j) = \{ \text{out}(i) | i \text{ is predecessor of } j \} \]

Note: \( \text{pres}(j) \) is the complement of \( \text{kill}(j) \), \( D - \text{kill}(j) \)

Note: What makes the 4 classical problems special is that sets \( \text{pres}(j) \) and \( \text{gen}(j) \) do not depend on \( \text{in}(j) \)

- Set union and set intersection can be implemented as logical OR and AND respectively