Announcements

- HW1 is posted, due January 30th
  - Will cover material this week and next week
- You can work individually or in teams of 2
  - Set up teams in Submitty
  - Ask questions on forum
  - Upload in Submitty
- Office hours: Wed Noon-2pm or by appt.
  - Steven’s office hours TBD

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. Determine the leader statements:
  (i) First program statement
  (ii) Targets of gotos, either conditional or unconditional
  (iii) Any statement following a goto

Step 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

Building the Control Flow Graph

Build the CFG from linear 3-address code:

- Step 1: partition code into basic blocks
  - Basic blocks are the nodes of the CFG
- Step 2: add control flow edges
Question. Find the Leader Statements

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

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.

Determine edges as follows:

(i) 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 \text{goto}.

(ii) There is an edge from \( B_1 \) to \( B_2 \) if there is a \text{goto} from the last statement in \( B_1 \) to the first statement in \( B_2 \).

Question. Add Control Flow Edges

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

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.

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

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. \( t6 = \text{prod} + t5 \)
7. \( \text{prod} = t6 \)
8. \( t7 = i + 1 \)
9. \( i = t7 \)
10. if \( i <= 20 \) goto 1
Question. Run Local Common Subexpression Elimination
1. \( t1 = 4 \)
2. \( t2 = a \[ t1 \] \)
3. \( t4 = b \[ t1 \] \)
4. \( t5 = t2 \times t4 \)
5. \( \text{prod} = \text{prod} + t5 \)
6. \( t1 = t1 + 4 \)
7. if \( t1 \leq 80 \) goto 2

Local Analysis: Dead Code Elimination
1. \( a = y+2 \) \quad (a,1)
2. \( z = x+w \) \quad (a,1),(z,2)
3. \( x = a \) \quad (a,1),(z,2),(x,3)
4. \( z = b+c \) \quad (a,1),(x,3),(z,4)
   \( z \) is redefined at 4, and was never used on the way from 2 to 4; thus \( 2 \cdot z=x+w \) is "dead code"
5. \( b = a \) \quad (a,1),(x,3),(z,4), (b,5)

After Local Common Subexpression and Dead Code Elimination
1. \( a = y + 2 \) \quad 1'. \( a = y + 2 \)
2. \( z = x + w \) \quad 2'. \( x = a \)
3. \( x = y + 2 \) \quad 3'. \( z = b + c \)
4. \( z = b + c \) \quad 4'. \( b = a \)
5. \( b = y + 2 \)

Local Constant Propagation
1. \( t1 = 1 \) \quad Assume a, k, t3, and t4 are used beyond basic block:
2. \( a = t1 \) \quad 1'. \( a = 1 \)
3. \( t2 = 1 + a \) \quad 2'. \( k = 2 \)
4. \( k = t2 \) \quad 3'. \( t4 = 8.2 \)
5. \( t3 = \text{cvttoreal}(k) \) \quad 4'. \( t3 = 8.2 \)
6. \( t4 = 6.2 + t3 \) \quad 5'. \( t3 = t4 \)

Arrays and Pointers Make Analysis 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 dataflow facts that hold along all execution paths for all inputs
  - Control splits and control merges
  - Loops (control goes back)
- Dataflow analysis is a powerful framework
- We can define many different kinds of dataflow analysis

---

Four Classical Dataflow Problems

- Reaching definitions (Reach)
- Live uses of variables (Live)
- Available expressions (Avail)
- Very busy expressions (VeryB)
- Reach and the dual Live enable several classical optimizations such as dead code elimination, as well as dataflow-based testing
- Avail enables global common subexpression elimination
- VeryB enables code motion

---

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., \( y=x+4 \))
- 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 Relations

- Use-def chain links a use of \( x \) to a definition of \( x \) that reaches that use
- Def-use chain links a definition to a use that it reaches
Def-use Enable Optimizations

- Dead code elimination (Def-use)
- Loop invariant 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

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

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

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

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

Applications of Def-use

- Secure computation
- Encryption inference
  - To encrypt data and transform programs
  - E.g., sums weight, BP over patients to compute average. Use of weight, BP is +, thus, use additively homomorphic encryption
- Protocol inference for multi-party computation
  - Different protocols implement operations at different costs

Applications of Def-use

- Vulnerability analysis and exploitation; e.g., info leaks

```c
void *fp = &exit;
printf("libc function @ %p\n" fp):
--- 0x7f2f2f8a497030
```
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)\}

Next, define the dataflow equations (i.e., effect of code at node \( j \) on incoming dataflow facts)

\[
\begin{align*}
    j: & \quad x = y + z \\
    \text{kill}(j): & \quad \text{all definitions of } (x, \_)
\end{align*}
\]

\[
\begin{align*}
    \text{gen}(j): & \quad \text{this definition of } (x, j) \\
    \text{out}(j): & \quad (\text{in}(j) - \text{kill}(j)) \cup \text{gen}(j)
\end{align*}
\]

E.g., if \( \text{in}(4) = \{(x, 1), (y, 2), (x, 3)\} \)
Node 4 is: \( x = y + z \)
Then \( \text{out}(4) = \{(y, 2), (x, 4)\} \)

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

For Reach, \( V \) is the set union \( \cup \)

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

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

Reach: Solution of Equations

\[
\begin{align*}
    \text{in}(1) = \emptyset & \quad \text{out}(1) = \{(x, 1)\} \\
    \text{in}(2) = \{(x, 1), (y, 2)\} & \quad \text{out}(2) = \{(x, 1), (y, 2)\} \\
    \text{in}(3) = \{(x, 1), (x, 5), (y, 2), (y, 4)\} & \quad \text{out}(3) = \{(x, 3)\} \\
    \text{in}(4) = \{(x, 1), (x, 5), (y, 2), (y, 4)\} & \quad \text{out}(4) = \{\{(x, 1), (x, 5), (y, 4)\} \}
\end{align*}
\]

Reach: Dataflow Equations

\[
\begin{align*}
    \text{in}(1) = \emptyset & \quad \text{out}(1) = \{(x, 1)\} \\
    \text{in}(2) = \{(x, 1), (y, 2)\} & \quad \text{out}(2) = \{(x, 1), (y, 2)\} \\
    \text{in}(3) = \{(x, 1), (x, 5), (y, 2), (y, 4)\} & \quad \text{out}(3) = \{(x, 3)\} \\
    \text{in}(4) = \{(x, 1), (x, 5), (y, 2), (y, 4)\} & \quad \text{out}(4) = \{\{(x, 1), (x, 5), (y, 4)\} \}
\end{align*}
\]

Reachings Definitions

\[
in(1) \quad \text{in}(2) \quad \text{in}(3) \\
\begin{array}{c}
\text{(1)} \\
\text{(2)} \\
\text{(3)}
\end{array}
\]

Forward, may dataflow problem
Problem 2. 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. \( x=2 \); 2. \( y=4 \); 3. \( x=1 \); if \( y>x \) then 5. \( z=y \); else 6. \( z=y*y \); 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*y
7. x=z
```

Live Uses of Variables (Live)

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

\[
in_{LV}(j) = (out_{LV}(j) \setminus kill_{LV}(j)) \cup gen_{LV}(j)
\]

\[
out_{LV}(j) = \{ \cup in_{LV}(i) \mid i \text{ is a successor of } j \}
\]

Q: What are the primitive dataflow facts?
Q: What is \( gen_{LV}(j) \)?
Q: What is \( kill_{LV}(j) \)?

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 \( n \)

\[
x \circ y, x = \cdots, y = \cdots
\]

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 \( j \): \( x = y \circ z \)?
  - \( \text{out}_{\text{AE}}(j) = (\text{in}_{\text{AE}}(j) \setminus \text{kill}_{\text{AE}}(j)) \cup \text{gen}_{\text{AE}}(j) \)
  - \( \text{kill}_{\text{AE}}(j) \): all expressions with operand \( x \): \( (x \circ \_), (\_ \circ x) \)
  - \( \text{gen}_{\text{AE}}(j) \): new expression: \( \{ (y \circ z) \} \)

Available Expressions (Avail)

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

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

Available Expressions (Avail)

Forward, must dataflow problem
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

Very Busy Expressions

- An expression \( x \ op \ y \) is very busy at node \( n \) if along EVERY path from \( n \) to the end of the program, we come to a computation of \( x \ op \ y \) 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 \( \text{gen}(i) \)?
Q: What is \( \text{kill}(i) \)?
Q: What is the merge operator?

Very Busy Expressions (VeryB)

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

Very Busy Expressions (VeryB)

- Merge operator?
  - For VeryB, it is set intersection \( \cap \)
  
  \[ \text{out}_{\text{vb}}(j) = \{ \bigcap \text{in}_{\text{vb}}(i) \mid i \text{ is a successor of } j \} \]

Very Busy Expressions

Backward, must dataflow problem

\[ j \]
\[ \text{out}_{\text{vb}}(j) \]
\[ \text{out}_{\text{vb}}(1) \]
\[ \text{out}_{\text{vb}}(2) \]
\[ \text{out}_{\text{vb}}(3) \]
## 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 \(n\))

### Similarities

- Dataflow equations (i.e., transfer functions) for forward problems have generic form:
  
  \[
  \text{out}(j) = F_j(\text{in}(j)) = (\text{in}(j) \setminus \text{kill}(j)) \cup \text{gen}(j) = \left(\text{in}(j) \cap \text{pres}(j)\right) \cup \text{gen}(j)
  \]

  \[
  \text{in}(j) = \{ \forall \text{out}(i) \mid i \text{ is predecessor of } j \}\}
  
  Note: pres\(j\) is the complement of kill\(j\), \(D \setminus \text{kill}(j)\)

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

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