**Announcements**

- HW1 is posted due January 30th
- You can work individually or in teams of 2
  - Setup teams in Submitty
- Bring a printout of homework on the due date
- Also, upload in Submitty

---

**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**

Build the CFG from linear 3-address code:

1. Step 1: partition code into basic blocks
   - Basic blocks are the nodes in the CFG
2. Step 2: add control flow edges

Aside: in Principles of Software, we built a CFG from structural IR:

\[
S ::= x = y \text{ Op } z \mid \text{if } (B) \text{ then } S \text{ else } S \mid \text{while } (B) \text{ S } \mid S ; S
\]
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. \( \text{sum} = 0 \)
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 sequence.

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 goto
   (ii) 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 \( B_2 \)

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 on a 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 the 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. \( t_1 = 4 \times i \) 
2. \( t_2 = a[t_1] \) 
3. \( t_3 = 4 \times i \) 
4. \( t_4 = b[t_3] \) 
5. \( t_5 = t_2 \times t_4 \) 
6. \( t_6 = \text{prod} \times t_5 \) 
7. \( \text{prod} = t_6 \) 
8. \( t_7 = i + 1 \) 
9. \( i = t_7 \) 
10. if \( i \leq 20 \) goto 1

Local Analysis: Dead Code Elimination

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

After Local Common Subexpression and Dead Code Elimination

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

Local Constant Propagation

1. \( t_1 = 1 \) Assume \( a, k, t_3, \) and \( t_4 \) are used beyond basic block: 
2. \( a = t_1 \) 1'. \( a = 1 \) 
3. \( t_2 = t_1 + a \) 2'. \( k = 2 \) 
4. \( k = t_2 \) 3'. \( t_4 = 8.2 \) 
5. \( t_3 = \text{cttopreal}(k) \) 4'. \( t_3 = 8.2 \) 
6. \( t_4 = 6.2 + t_3 \) 
7. \( t_3 = t_4 \) 

David Gries’ algorithm:
- Process 3-address statements in order
- Check if operand is constant; if so, substitute
- If all operands are constant:
  - Do operation, and add (LHS,value) to map
- If not all operands constant:
  - Delete (LHS,value) entry from map

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
- Not as “powerful” as Hoare logic

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 conservative 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., \( x \) 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 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)
- 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

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

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

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

“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., { (1,1), (p,4) }
Reaching Definitions (Reach)

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

$$j: x = y+z \quad \text{kill(j): all definitions of } (x,_)\quad \text{gen(j): this definition of } x, (x,j)$$

$$\text{out(j)} = (\text{in(j)} - \text{kill(j)}) \cup \text{gen(j)}$$

E.g., if $\text{in(4)} = \{(x,1), (y,2), (x,3)\}$

Node 4 is: $x = y+z$

Then $\text{out(4)} = \{(y,2), (x,4)\}$

Spring 18 CSCI 4450/6450, A. Milanova

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 Uses of Variables (Live)

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

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

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

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

Live Example

![Diagram of a control flow graph with variables and operations]

Live Uses of Variables

- Data
  - Primitive facts: variables \( x \)
  - Propagates sets: \( \{ x, y, z \} \)

- Dataflow equations. At \( j: x = y + z \)
  - \( kill_{LV}(j): x \)
  - \( gen_{LV}(j): \{ y, z \} \)

- 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**

- $z = a \times b$
- $r = 2 \times z$
- $u = a \times b$
- $z = u / 2$

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

**Merge operator?**

- For Avail, it is set intersection $\bigcap$

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

**Example**

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

\[ \begin{array}{c}
\text{Variable declarations} \\
X = \ldots \\
Y = \ldots \\
t1 = X \ op \ Y \\
X = \ldots \\
Y = \ldots \\
t1 = X \ op \ Y \\
\end{array} \]

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 \( \text{gen}_{\text{VB}}(i) \)?
- Q: What is \( \text{kill}_{\text{VB}}(i) \)?
- Q: What is the merge operator?

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

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

\[ \text{out}_{\text{VB}}(j) = \bigcap \{ \text{in}_{\text{VB}}(i) \mid i \text{ is successor of } j \} \]

Dataflow Analysis Problems

<table>
<thead>
<tr>
<th>May Analyses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Analyses</td>
</tr>
<tr>
<td>Reaching Definitions</td>
</tr>
<tr>
<td>Available Expressions</td>
</tr>
<tr>
<td>Backward Analyses</td>
</tr>
<tr>
<td>Live Uses of Variables</td>
</tr>
<tr>
<td>Very Busy Expressions</td>
</tr>
</tbody>
</table>
Similarities

- In all cases, analysis operates on a finite set \( D \) of primitive dataflow facts:
  - \textit{Reach}: \( D \) is the set of all definitions in the program:
    \( e.g., \{(x,1), (y,2), (x,4), (y,5)\} \)
  - \textit{Avail} and \textit{VeryB}: \( D \) is the set of all arithmetic expressions:
    \( e.g., \{a+b, a*b, a+1\} \)
  - \textit{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 \( n \))

Spring 18 CSCI-4450/6450, A. Milanova