CSC D70: Compiler Optimization Dataflow Analysis

Prof. Gennady Pekhimenko
University of Toronto
Winter 2019

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
Refreshing from Last Lecture

• Basic Block Formation

• Value Numbering
Partitioning into Basic Blocks

• Identify the leader of each basic block
  – First instruction
  – Any target of a jump
  – Any instruction immediately following a jump

• Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)
1) \( i = 1 \)
2) \( j = 1 \)
3) \( t_1 = 10 \cdot i \)
4) \( t_2 = t_1 + j \)
5) \( t_3 = 8 \cdot t_2 \)
6) \( t_4 = t_3 - 88 \)
7) \( a[t_4] = 0.0 \)
8) \( j = j + 1 \)
9) if \( j \leq 10 \) goto (3)
10) \( i = i + 1 \)
11) if \( i \leq 10 \) goto (2)
12) \( i = 1 \)
13) \( t_5 = i - 1 \)
14) \( t_6 = 88 \cdot t_5 \)
15) \( a[t_6] = 1.0 \)
16) \( i = i + 1 \)
17) if \( i \leq 10 \) goto (13)

\( = \text{Leader} \)
Value Numbering (VN)

• More explicit with respect to VALUES, and TIME

• each value has its own “number”
  – common subexpression means same value number

• var2value: current map of variable to value
  – used to determine the value number of current expression

\[ r1 + r2 \Rightarrow \text{var2value}(r1)+\text{var2value}(r2) \]
Algorithm

Data structure:
VALUES = Table of
expression     // [OP, valnum1, valnum2]
var            // name of variable currently holding expression

For each instruction (dst = src1 OP src2) in execution order

valnum1 = var2value(src1); valnum2 = var2value(src2);

IF [OP, valnum1, valnum2] is in VALUES
  v = the index of expression
  Replace instruction with CPY dst = VALUES[v].var
ELSE
  Add
  expression = [OP, valnum1, valnum2]
  var        = dst
  to VALUES
  v = index of new entry; tv is new temporary for v
  Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                          CPY dst = tv;

set_var2value (dst, v)
VN Example

Assign: a→r1, b→r2, c→r3, d→r4

\[ a = b+c; \quad \text{ADD } t1 = r2,r3 \]

\[ \text{CPY } r1 = t1 \quad // (a = t1) \]

\[ b = a-d; \quad \text{SUB } t2 = r1,r4 \]

\[ \text{CPY } r2 = t2 \quad // (b = t2) \]

\[ c = b+c; \quad \text{ADD } t3 = r2,r3 \]

\[ \text{CPY } r3 = t3 \quad // (c = t3) \]

\[ d = a-d; \quad \text{CPY } r2 = t2 \]
Questions about Assignment #1

• Tutorial #1

• Tutorial #2 next week
  – More in-depth LLVM coverage
Outline

1. Structure of data flow analysis
2. Example 1: Reaching definition analysis
3. Example 2: Liveness analysis
4. Generalization
What is Data Flow Analysis?

• **Local analysis (e.g., value numbering)**
  – analyze effect of each instruction
  – compose effects of instructions to derive information from beginning of basic block to each instruction

• **Data flow analysis**
  – analyze effect of each basic block
  – compose effects of basic blocks to derive information at basic block boundaries
  – from basic block boundaries, apply local technique to generate information on instructions
What is Data Flow Analysis? (2)

• Data flow analysis:
  – Flow-sensitive: sensitive to the control flow in a function
  – intraprocedural analysis

• Examples of optimizations:
  – Constant propagation
  – Common subexpression elimination
  – Dead code elimination
What is Data Flow Analysis? (3)

For each variable x determine:
Value of x?
Which “definition” defines x?
Is the definition still meaningful (live)?
Static Program vs. Dynamic Execution

- **Statically**: Finite program
- **Dynamically**: Can have infinitely many possible execution paths
- **Data flow analysis abstraction**: For each point in the program:
  - combines information of all the instances of the same program point.
- **Example of a data flow question**:
  - Which definition defines the value used in statement “b = a”?
Effects of a Basic Block

• Effect of a statement: \( a = b + c \)
  • \textbf{Uses} variables (b, c)
  • \textbf{Kills} an old definition (old definition of a)
  • new \textbf{definition} (a)

• Compose effects of statements \(-\) Effect of a basic block
  – A \textbf{locally exposed use} in a b.b. is a use of a data item which is not preceded in the b.b. by a definition of the data item
  – any definition of a data item in the basic block \textbf{kills} all definitions of the same data item reaching the basic block.
  – A \textbf{locally available definition} = last definition of data item in b.b.
Effects of a Basic Block

A **locally available definition** = last definition of data item in b.b.

\[
\begin{align*}
t_1 &= r_1 + r_2 \\
r_2 &= t_1 \\
t_2 &= r_2 + r_1 \\
r_1 &= t_2 \\
t_3 &= r_1 \times r_1 \\
r_2 &= t_3 \\
\text{if } r_2 > 100 \text{ goto } L1
\end{align*}
\]

Locally exposed uses? $r_1$

Kills any definitions? Any other definition of $t_2$

Locally avail. definition? $t_2$
Reaching Definitions

• Every assignment is a **definition**
• A **definition** *d* **reaches** a point *p* 
  if **there exists** path from the point immediately following *d* to *p* such that *d* is **not killed** (overwritten) along that path.
• Problem statement
  – For each point in the program, determine if each definition in the program reaches the point
  – A bit vector per program point, vector-length = #defs
Reaching Definitions (2)

Every assignment is a definition

A definition \(d\) reaches a point \(p\) if there exists path from the point immediately following \(d\) to \(p\) such that \(d\) is not killed (overwritten) along that path.

Problem statement

- For each point in the program, determine if each definition in the program reaches the point
- A bit vector per program point, vector-length = \#defs

- d2 reaches this point?
Reaching Definitions (3)

L1: if input() GOTO L2

d0: a = x

d1: b = a
d2: a = y
GOTO L1

L2: ...

d2 reaches this point?

yes
Data Flow Analysis Schema

- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between in[b] and out[b] for all basic blocks b
  - Effect of code in basic block:
    - Transfer function $f_b$ relates in[b] and out[b], for same b
  - Effect of flow of control:
    - relates out[b_1], in[b_2] if b_1 and b_2 are adjacent
- Find a solution to the equations
Effects of a Statement

\[ \text{in}[B0] \]

\[ \begin{align*}
\text{d0: } & y = 3 & f_{d0} \\
\text{d1: } & x = 10 & f_{d1} \\
\text{d2: } & y = 11 & f_{d2}
\end{align*} \]

\[ \text{out}[B0] \]

- \( f_s \): A transfer function of a statement
  - abstracts the execution with respect to the problem of interest
- For a statement \( s \) (\( d: x = y + z \))

\[ \text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s]-\text{Kill}[s]) \]

  - \( \text{Gen}[s] \): definitions generated: \( \text{Gen}[s] = \{d\} \)
  - \( \text{Propagated} \) definitions: \( \text{in}[s] - \text{Kill}[s] \),
    where \( \text{Kill}[s] \)=set of all other defs to \( x \) in the rest of program
Effects of a Basic Block

- Transfer function of a statement $s$:
  - $\text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s]-\text{Kill}[s])$

- Transfer function of a basic block $B$:
  - Composition of transfer functions of statements in $B$
  - $\text{out}[B] = f_B(\text{in}[B]) = f_{d_2} \cdot f_{d_1} \cdot f_{d_0}(\text{in}[B])$
    - $= \text{Gen}[d_2] \cup (\text{Gen}[d_1] \cup (\text{Gen}[d_0] \cup (\text{in}[B]-\text{Kill}[d_0]))-\text{Kill}[d_1]))-\text{Kill}[d_2]$ 
    - $= \text{Gen}[d_2] \cup (\text{Gen}[d_1] \cup (\text{Gen}[d_0] - \text{Kill}[d_1]) - \text{Kill}[d_2]) \cup \text{in}[B] - (\text{Kill}[d_0] \cup \text{Kill}[d_1] \cup \text{Kill}[d_2])$
    - $= \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B])$
  - $\text{Gen}[B]$: locally exposed definitions (available at end of bb)
  - $\text{Kill}[B]$: set of definitions killed by $B$
Example

- a **transfer function** $f_b$ of a basic block $b$:
  \[
  \text{OUT}[b] = f_b(\text{IN}[b])
  \]
  incoming reaching definitions $\rightarrow$ outgoing reaching definitions

- A basic block $b$
  - generates definitions: $\text{Gen}[b]$,
    - set of locally available definitions in $b$
  - kills definitions: $\text{in}[b] - \text{Kill}[b]$,
    where $\text{Kill}[b]$=set of defs (in rest of program) killed by defs in $b$

- $\text{out}[b] = \text{Gen}[b] \cup (\text{in}(b)-\text{Kill}[b])$
Effects of the Edges (acyclic)

- \( \text{out}[b] = f_b(\text{in}[b]) \)
- Join node: a node with multiple predecessors
- **meet** operator:
  \[ \text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \ldots \cup \text{out}[p_n], \text{where } p_1, \ldots, p_n \text{ are all predecessors of } b \]
Example

- out[b] = f_b(in[b])
- Join node: a node with multiple predecessors
- **meet** operator:
  \[ \text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup ... \cup \text{out}[p_n], \text{where} \]
  \[ p_1, ..., p_n \text{ are all predecessors of } b \]
Cyclic Graphs

• Equations still hold
  • \(\text{out}[b] = f_b(\text{in}[b])\)
  • \(\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup ... \cup \text{out}[p_n], p_1, ..., p_n \text{ pred.}\)

• Find: fixed point solution
Reaching Definitions: Iterative Algorithm

input: control flow graph $\text{CFG} = (N, E, \text{Entry}, \text{Exit})$

// Boundary condition
out[Entry] = ∅

// Initialization for iterative algorithm
For each basic block $B$ other than Entry
out[B] = ∅

// iterate
While (Changes to any out[] occur) {
   For each basic block $B$ other than Entry {
      in[B] = $\bigcup$ (out[p]), for all predecessors $p$ of $B$
   }
}
Reaching Definitions: Worklist Algorithm

input: control flow graph $CFG = (N, E, \text{Entry}, \text{Exit})$

// Initialize
\[
\text{out}[\text{Entry}] = \emptyset \quad \text{// can set out[Entry] to special def}
\]
\[
\text{out}[i] = \emptyset \quad \text{// if reaching then undefined use}
\]
For all nodes $i$
\[
\text{out}[i] = \emptyset \quad \text{// can optimize by out[i]=gen[i]}
\]
\[
\text{ChangedNodes} = N
\]

// iterate

While $\text{ChangedNodes} \neq \emptyset$ {
\[
\text{Remove } i \text{ from ChangedNodes}
\]
\[
\text{in}[i] = U(\text{out}[p]), \text{ for all predecessors } p \text{ of } i
\]
\[
\text{oldout} = \text{out}[i]
\]
\[
\text{out}[i] = f_i(\text{in}[i]) \quad \text{// out[i]=gen[i]U(in[i]-kill[i])}
\]
\[
\text{if (oldout} \neq \text{out}[i]) {
\quad \text{for all successors } s \text{ of } i
\quad \text{add } s \text{ to ChangedNodes}
\}
\]
Example

B1
- d1: i = n
- d2: j = n
- d3: a = u1

B2
- d4: i = i + 1
- d5: j = j - 1

B3
- d6: a = u2

B4
- d7: i = u3

<table>
<thead>
<tr>
<th></th>
<th>First Pass</th>
<th>Second Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN[B1]</td>
<td>000 00 0 0</td>
<td>000 00 0 0</td>
</tr>
<tr>
<td>OUT[B1]</td>
<td>111 00 0 0</td>
<td>111 00 0 0</td>
</tr>
<tr>
<td>IN[B2]</td>
<td>111 00 0 0</td>
<td>111 01 1 1</td>
</tr>
<tr>
<td>OUT[B2]</td>
<td>001 11 0 0</td>
<td>001 11 1 0</td>
</tr>
<tr>
<td>IN[B3]</td>
<td>001 11 0 0</td>
<td>001 11 1 0</td>
</tr>
<tr>
<td>OUT[B3]</td>
<td>000 11 1 0</td>
<td>000 11 1 0</td>
</tr>
<tr>
<td>IN[B4]</td>
<td>001 11 1 0</td>
<td>001 11 1 0</td>
</tr>
<tr>
<td>OUT[B4]</td>
<td>001 01 1 1</td>
<td>001 01 1 1</td>
</tr>
<tr>
<td>IN[exit]</td>
<td>001 01 1 1</td>
<td>001 01 1 1</td>
</tr>
</tbody>
</table>
Live Variable Analysis

• Definition
  – A variable $v$ is live at point $p$ if
    • the value of $v$ is used along some path in the flow graph starting at $p$.
  – Otherwise, the variable is dead.

• Motivation
  • e.g. register allocation
    
    ```
    for i = 0 to n
      ... i ...
    ... 
    for i = 0 to n
      ... i ...
    ```

• Problem statement
  – For each basic block
    • determine if each variable is live in each basic block
  – Size of bit vector: one bit for each variable
Transfer Function

• **Insight:** Trace uses backwards to the definitions
  an execution path control flow example

```
def
IN[b] = f_b(OUT[b])
```
d3: a = 1
d4: b = 1

d5: c = a
d6: a = 4

• **A basic block b can**
  • **generate** live variables: $\text{Use}[b]$
    – set of locally exposed uses in $b$
  • **propagate** incoming live variables: $\text{OUT}[b] - \text{Def}[b]$
    – where $\text{Def}[b]$ = set of variables defined in $b$

• **transfer function** for block $b$:
  $\text{in}[b] = \text{Use}[b] \cup (\text{out}(b) - \text{Def}[b])$
• $\text{in}[b] = f_b(\text{out}[b])$
• **Join node**: a node with multiple successors
• **meet** operator:
  $$\text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup ... \cup \text{in}[s_n],$$
  where
  $$s_1, ..., s_n$$ are all successors of $b$
• in[b] = f_b (out[b])
• Join node: a node with multiple successors
• meet operator:
  out[b] = in[s_1] U in[s_2] U ... U in[s_n], where 
  s_1, ..., s_n are all successors of b
Liveness: Iterative Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
in[Exit] = ∅

// Initialization for iterative algorithm
For each basic block B other than Exit
  in[B] = ∅

// iterate
While (Changes to any in[] occur) {
  For each basic block B other than Exit {
    out[B] = U (in[s]), for all successors s of B
    in[B] = f_B(out[B])  // in[B]=Use[B] U (out[B]-Def[B])
  }
}
Example

- **B1**
  - \( d_1: i = n + 1 \)
  - \( d_2: i = n \)
  - \( d_3: a = u_2 \)

- **B2**
  - \( d_4: i = i + 1 \)
  - \( d_5: j = j - 1 \)

- **B3**
  - \( d_6: a = u_2 \)

- **B4**
  - \( d_7: i = u_3 \)

**First Pass**
- OUT[entry] \( \{m,n,u_1,u_2,u_3\} \)
- IN[B1] \( \{m,n,u_1,u_2,u_3\} \)
- OUT[B1] \( \{i,j,u_2,u_3\} \)
- IN[B2] \( \{i,j,u_2,u_3\} \)
- OUT[B2] \( \{u_2,u_3\} \)
- IN[B3] \( \{u_2,u_3\} \)
- OUT[B3] \( \{u_3\} \)
- IN[B4] \( \{u_3\} \)
- OUT[B4] \( \{\} \)

**Second Pass**
- OUT[entry] \( \{m,n,u_1,u_2,u_3\} \)
- IN[B1] \( \{m,n,u_1,u_2,u_3\} \)
- OUT[B1] \( \{i,j,u_2,u_3\} \)
- IN[B2] \( \{i,j,u_2,u_3\} \)
- OUT[B2] \( \{u_2,u_3\} \)
- IN[B3] \( \{u_2,u_3\} \)
- OUT[B3] \( \{u_3\} \)
- IN[B4] \( \{u_3\} \)
- OUT[B4] \( \{i,j,u_2,u_3\} \)
## Framework

<table>
<thead>
<tr>
<th>Domain</th>
<th>Reaching Definitions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forward:</td>
<td>out(_b) = f(_b)(in(_b))</td>
<td>back:</td>
</tr>
<tr>
<td></td>
<td>in(_b) = ∧ out[\text{pred}(b)]</td>
<td>in(_b) = f(_b)(out(_b))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>out(_b) = ∧ in[\text{succ}(b)]</td>
</tr>
<tr>
<td>Transfer function</td>
<td>f(_b)(x) = \text{Gen}_b \cup (x - \text{Kill}_b)</td>
<td>f(_b)(x) = \text{Use}_b \cup (x - \text{Def}_b)</td>
</tr>
<tr>
<td>Meet Operation (\∧)</td>
<td>\cup</td>
<td>\cup</td>
</tr>
<tr>
<td>Boundary Condition</td>
<td>out[entry] = ∅</td>
<td>in[exit] = ∅</td>
</tr>
<tr>
<td>Initial interior points</td>
<td>out(_b) = ∅</td>
<td>in(_b) = ∅</td>
</tr>
</tbody>
</table>

Other examples (e.g., Available expressions), defined in ALSU 9.2.6
Thought Problem 1. “Must-Reach” Definitions

• A definition $D$ ($a = b+c$) must reach point $P$ iff
  – $D$ appears at least once along on all paths leading to $P$
  – $a$ is not redefined along any path after last appearance of $D$ and before $P$

• How do we formulate the data flow algorithm for this problem?
Thought Problem 2: A legal solution to (May) Reaching Def?

- Will the worklist algorithm generate this answer?
Questions

• **Correctness**
  - equations are satisfied, if the program terminates.

• **Precision: how good is the answer?**
  - is the answer ONLY a union of all possible executions?

• **Convergence: will the analysis terminate?**
  - or, will there always be some nodes that change?

• **Speed: how fast is the convergence?**
  - how many times will we visit each node?
Foundations of Data Flow Analysis

1. Meet operator
2. Transfer functions
3. Correctness, Precision, Convergence
4. Efficiency

• Reference: ALSU pp. 613-631
• Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
A Unified Framework

• Data flow problems are defined by
  • Domain of values: \( V \)
  • Meet operator \((V \land V \to V)\), initial value
  • A set of transfer functions \((V \to V)\)

• Usefulness of unified framework
  • To answer questions such as correctness, precision, convergence, speed of convergence for a family of problems
    – If meet operators and transfer functions have properties \( X \), then we know \( Y \) about the above.
  • Reuse code
Meet Operator

• Properties of the meet operator
  • commutative: \( x \land y = y \land x \)
  • idempotent: \( x \land x = x \)
  • associative: \( x \land (y \land z) = (x \land y) \land z \)
  • there is a Top element \( T \) such that \( x \land T = x \)

• Meet operator defines a partial ordering on values
  • \( x \leq y \) if and only if \( x \land y = x \) (\( y \rightarrow x \) in diagram)
    – Transitivity: if \( x \leq y \) and \( y \leq z \) then \( x \leq z \)
    – Antisymmetry: if \( x \leq y \) and \( y \leq x \) then \( x = y \)
    – Reflexitivity: \( x \leq x \)
Partial Order

- Example: let $\mathbf{V} = \{x \mid \text{such that } x \subseteq \{d_1, d_2\}\}$, $\land = \cap$

- Top and Bottom elements
  - Top $T$ such that: $x \land T = x$
  - Bottom $\perp$ such that: $x \land \perp = \perp$

- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a $T$, but not necessarily a $\perp$.
- $x, y$ are ordered: $x \leq y$ then $x \land y = x$ (y -> x in diagram)
- what if $x$ and $y$ are not ordered?
  - $x \land y \leq x, x \land y \leq y$, and if $w \leq x, w \leq y$, then $w \leq x \land y$
One vs. All Variables/Definitions

• Lattice for each variable: e.g. intersection

\[
\begin{array}{c}
1 \\
\downarrow \\
0 \\
\end{array}
\]

• Lattice for three variables:
Descending Chain

• **Definition**
  - The **height** of a lattice is the largest number of > relations that will fit in a descending chain.
  
    \[ x_0 > x_1 > x_2 > \ldots \]

• **Height of values in reaching definitions?**
  - Height n – number of definitions

• **Important property:** **finite descending chain**

• **Can an infinite lattice have a finite descending chain?**
  - yes

• **Example: Constant Propagation/Folding**
  - To determine if a variable is a constant

• **Data values**
  - undef, ... -1, 0, 1, 2, ..., not-a-constant
Transfer Functions

• Basic Properties \( f: V \rightarrow V \)
  – Has an identity function
    • There exists an \( f \) such that \( f(x) = x \), for all \( x \).
  – Closed under composition
    • if \( f_1, f_2 \in F \), then \( f_1 \cdot f_2 \in F \)
Monotonicity

• A framework \((F, V, \land)\) is monotone if and only if
  • \(x \leq y\) implies \(f(x) \leq f(y)\)
  • i.e. a “smaller or equal” input to the same function will always give a “smaller or equal” output

• Equivalently, a framework \((F, V, \land)\) is monotone if and only if
  • \(f(x \land y) \leq f(x) \land f(y)\)
  • i.e. merge input, then apply \(f\) is small than or equal to apply the transfer function individually and then merge the result
Example

• Reaching definitions: \( f(x) = \text{Gen} \cup (x - \text{Kill}), \land = \lor \)
  
  – Definition 1:
  
  \( x_1 \leq x_2, \text{Gen} \cup (x_1 - \text{Kill}) \leq \text{Gen} \cup (x_2 - \text{Kill}) \)
  
  – Definition 2:
  
  \( (\text{Gen} \cup (x_1 - \text{Kill})) \cup (\text{Gen} \cup (x_2 - \text{Kill})) = (\text{Gen} \cup ((x_1 \cup x_2) - \text{Kill})) \)

• Note: Monotone framework does not mean that \( f(x) \leq x \)
  
  • e.g., reaching definition for two definitions in program
  • suppose: \( f_x: \text{Gen}_x = \{d_1, d_2\}; \text{Kill}_x = {} \)

• If input(second iteration) \( \leq \) input(first iteration)
  
  • result(second iteration) \( \leq \) result(first iteration)
Distributivity

• A framework $(F, V, \wedge)$ is **distributive** if and only if
  • $f(x \wedge y) = f(x) \wedge f(y)$
  • i.e. merge input, then apply $f$ is **equal to** apply the transfer function individually then merge result

• Example: Constant Propagation is NOT distributive

\[
\begin{align*}
  a &= 2 \\
  b &= 3 \\
\end{align*}
\]

\[
\begin{align*}
  a &= 3 \\
  b &= 2 \\
\end{align*}
\]

\[
c = a + b
\]
Data Flow Analysis

• Definition
  – Let $f_1, ..., f_m : \in F$, where $f_i$ is the transfer function for node $i$
    • $f_p = f_{n_k} \cdot ... \cdot f_{n_1}$, where $p$ is a path through nodes $n_1, ..., n_k$
    • $f_p = \text{identify function}$, if $p$ is an empty path

• Ideal data flow answer:
  – For each node $n$:
    $\bigwedge f_{p_i}(T)$, for all possibly executed paths $p_i$ reaching $n$.

• But determining all possibly executed paths is undecidable
Meet-Over-Paths (MOP)

- Error in the conservative direction
- **Meet-Over-Paths** (MOP):
  - For each node $n$:
    \[
    \text{MOP}(n) = \bigwedge f_{p_i}(T), \text{ for all paths } p_i \text{ reaching } n
    \]
  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - MOP = Perfect-Solution $\bigwedge$ Solution-to-Unexecuted-Paths
  - MOP $\leq$ Perfect-Solution
  - Potentially more constrained, solution is small
    - hence *conservative*
  - It is not *safe* to be $>$ Perfect-Solution!
- **Desirable solution: as close to MOP as possible**
MOP Example

Assume: B2 & B3 do not update x

Ideal: Considers only 2 paths
B1-B2-B4-B6-B7 (i.e., x=1)
B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths
B1-B2-B4-B5-B7
B1-B3-B4-B6-B7
Solving Data Flow Equations

• **Example: Reaching definitions**
  • $\text{out[entry]} = \{\}$
  • Values = \{subsets of definitions\}
  • Meet operator: $\bigcup$
    • $\text{in[b]} = \bigcup \text{out}[p]$, for all predecessors $p$ of $b$
  • **Transfer functions**: $\text{out[b]} = \text{gen}_b \bigcup (\text{in[b]} - \text{kill}_b)$

• **Any solution satisfying equations** = **Fixed Point Solution (FP)**

• **Iterative algorithm**
  • initializes $\text{out[b]}$ to $\{\}$
  • if converges, then it computes **Maximum Fixed Point (MFP)**:
    • MFP is the largest of all solutions to equations

• **Properties**: 
  • $\text{FP} \leq \text{MFP} \leq \text{MOP} \leq \text{Perfect-solution}$
  • FP, MFP are safe
  • $\text{in}(b) \leq \text{MOP}(b)$
Partial Correctness of Algorithm

• If data flow framework is monotone, then if the algorithm converges, $\text{IN}[b] \leq \text{MOP}[b]$

• Proof: Induction on path lengths

  – Define $\text{IN}[\text{entry}] = \text{OUT}[\text{entry}]$
    and transfer function of entry = Identity function

  – Base case: path of length 0
    • Proper initialization of $\text{IN}[\text{entry}]$

  – If true for path of length $k$, $p_k = (n_1, \ldots, n_k)$, then
    true for path of length $k+1$: $p_{k+1} = (n_1, \ldots, n_{k+1})$
    • Assume: $\text{IN}[n_k] \leq f_{nk-1}(f_{nk-2}(\ldots f_{n1}(\text{IN}[\text{entry}])))$

      • $\text{IN}[n_{k+1}] = \text{OUT}[n_k] \wedge \ldots$
        $\leq \text{OUT}[n_k]$
        $\leq f_{nk}(\text{IN}[n_k])$
        $\leq f_{nk-1}(f_{nk-2}(\ldots f_{n1}(\text{IN}[\text{entry}])))$
Precision

• If data flow framework is **distributive**, then if the algorithm converges, $\text{IN}[b] = \text{MOP}[b]$

\[
\begin{array}{c}
a = 2 \\
b = 3
\end{array} \quad \begin{array}{c}
a = 3 \\
b = 2
\end{array}
\]

\[c = a + b\]

• Monotone but not distributive: behaves as if there are additional paths
Additional Property to Guarantee Convergence

• Data flow framework (monotone) converges if there is a finite descending chain

• For each variable IN[b], OUT[b], consider the sequence of values set to each variable across iterations:
  
  – if sequence for in[b] is monotonically decreasing
    • sequence for out[b] is monotonically decreasing
      • (out[b] initialized to T)
  
  – if sequence for out[b] is monotonically decreasing
    • sequence of in[b] is monotonically decreasing
Speed of Convergence

• Speed of convergence depends on order of node visits

• Reverse “direction” for backward flow problems
Reverse Postorder

- **Step 1: depth-first post order**
  ```
  main() {
      count = 1;
      Visit(root);
  }
  
  Visit(n) {
      for each successor s that has not been visited
          Visit(s);
      PostOrder(n) = count;
      count = count+1;
  }
  ```

- **Step 2: reverse order**
  ```
  For each node i
  rPostOrder = NumNodes - PostOrder(i)
  ```
Depth-First Iterative Algorithm (forward)

input: control flow graph CFG = (N, E, Entry, Exit)
/* Initialize */

\[
\text{out}[\text{entry}] = \text{init\_value}
\]
For all nodes \( i \)
\[
\text{out}[i] = T
\]
Change = True
/* iterate */

While Change {
    Change = False
    For each node \( i \) in rPostOrder {
        \[
        \text{in}[i] = \land(\text{out}[p]), \text{for all predecessors } p \text{ of } i
        \]
        oldout = \text{out}[i]
        \[
        \text{out}[i] = f_i(\text{in}[i])
        \]
        if oldout \neq \text{out}[i]
            Change = True
    }
}
Speed of Convergence

• If cycles do not add information
  • information can flow in one pass down a series of nodes of increasing order number:
    • e.g., 1 -> 4 -> 5 -> 7 -> 2 -> 4 ...
  • passes determined by number of back edges in the path
    • essentially the nesting depth of the graph
  • Number of iterations = number of back edges in any acyclic path + 2
    • (2 are necessary even if there are no cycles)

• What is the depth?
  – corresponds to depth of intervals for “reducible” graphs
  – in real programs: average of 2.75
A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?

- **Transfer functions**
  - function of each basic block
  - monotone
  - distributive?

- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph
Conclusions

• Dataflow analysis examples
  – Reaching definitions
  – Live variables

• Dataflow formation definition
  – Meet operator
  – Transfer functions
  – Correctness, Precision, Convergence
  – Efficiency
CSC D70: Compiler Optimization Dataflow Analysis

Prof. Gennady Pekhimenko
University of Toronto
Winter 2019

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons