Outline of Today’s Class
- Catch up
- Dataflow frameworks
- Lattices
- Transfer functions
- Worklist algorithm

Reading:
- Dragon Book, Chapter 9.2 and 9.3

Dataflow Analysis: Dataflow Frameworks

Outline of Today’s Class
- Catch up
- Dataflow frameworks
- Lattices
- Transfer functions
- Worklist algorithm

Spring 19 CSCI 4450/6450, A Milanova

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

Problem 1: Reaching Definitions

1. $x=2$
2. $y=4$
3. $x=1$
4. $(y>x)$ then $5. z=y$
   else $6. z=y*y$
7. $x=z$

What variables are live on exit from statement 3? Statement 1?

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. $x=1$
4. $(y>x)$ 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
Problem statement: for each node \( n \), compute the set of variables that may be live on exit from \( n \).

\[
\begin{align*}
\text{in}_{LV}(j) &= (\text{out}_{LV}(j) \cup \text{gen}_{LV}(j)) \\
\text{out}_{LV}(j) &= \{ \cup \text{in}_{LV}(i) \mid i \text{ is a successor of } j \}
\end{align*}
\]

Q: What are the primitive dataflow facts?
Q: What is \( \text{gen}_{LV}(j) \)?
Q: What is \( \text{kill}_{LV}(j) \)?

Problem 3: Available Expressions (Avail)

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 \( y \).

Can we eliminate \( w=a*b \)?
Available Expressions (Avail)

- Data?
  - Primitive dataflow facts are expressions, e.g., x+y, a*b, a+2
  - Analysis propagates sets of expressions, e.g., \{x+y, a*b\}
- Dataflow equations at j: x = y op z?
  - out_{AE}(j) = (in_{AE}(j) - kill_{AE}(j)) \cup gen_{AE}(j)
  - kill_{AE}(j): all expressions with operand x:
    - (x op _), (_ op x)
  - gen_{AE}(j): new expression: \{(y op z)\}

Example

1. y=a+b
2. x=a*b
3. if y<=a*b
4. a=a+1
5. x=a*b
6. goto 3
7. ...

Problem 3: Available Expressions

- Merge operator?
  - For Avail, it is set intersection \( \bigcap \)
  \[ in_{AE}(j) = \bigcap \{ out_{AE}(i) \mid i \text{ is predecessor of } j \} \]

Problem 4: Very Busy Expressions (VeryB)

- 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 \( y \).

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 op z?
  - in_{VB}(j) = (out_{VB}(j) - kill_{VB}(j)) \cup gen_{VB}(j)
  - kill_{VB}(j): all expressions with operand x:
    - (x op _), (_ op x)
  - gen_{VB}(j): new expression: \{(y op z)\}
**Very Busy Expressions (VeryB)**

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

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

**Another Example: Taint Analysis**

- A definition \((x, k)\) is tainted if \(k\) is designated as a taint source, or \((x, k)\) is computed based on an operand that is tainted.

- Problem statement: for each node \(n\), compute the set of tainted definitions that may reach \(n\).

**Outline of Today’s Class**

- Catch up
- Dataflow frameworks
- Lattice
- Transfer functions
- Worklist algorithm

- Reading:
  - Dragon Book, Chapter 9.2 and 9.3

**Dataflow Problems**

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

- Analyses operate over similar property spaces.
- In all cases, analysis operates over a finite set \( D \) of primitive dataflow facts.
  - Reach: \( D \) is the set of all definitions in the program:
    
    \[ \{(x,1), (y,2), (x,4), (y,5)\} \]
  - Avail and VeryB: \( D \) is the set of all arithmetic expressions:
    
    \[ \{a+b, a*a, a+1\} \]
  - Live: \( D \) is the set of all variables
    
    \[ \{x, y, z\} \]

- Solution at node \( n \) is a subset of \( D \) (e.g., a definition either reaches \( n \) or it does not reach \( n \)).

Dataflow Frameworks

- We generalize and study the properties of the property space.
  - Property space is a lattice.
  - Choice settles merge operator.
- We generalize and study the properties of the transfer function space.
  - Functions are monotone or distributive.
- We generalize and study the properties of the worklist algorithm that computes a solution.

Lattice Theory

- Partial ordering (denoted by \( \leq \) or \( \sqsubseteq \)):
  - Relation between pairs of elements
  - Reflexive \( a \leq a \)
  - Anti-symmetric \( a \leq b \) and \( b \leq a \implies a = b \)
  - Transitive \( a \leq b \) and \( b \leq c \implies a \leq c \)
- Partially ordered set (poset) (set \( S \), \( \leq \)):
  - 0 Element \( 0 \leq a \), for every \( a \) in \( S \)
  - 1 Element \( a \leq 1 \), for every \( a \) in \( S \)

We don’t necessarily need 0 and 1 element.

Poset Example

\[ D = \{a, b, c\} \]

The poset is \( 2^3 \), \( \leq \) is set inclusion.
Lattice Theory

- Greatest lower bound (glb)
  - \( l_1, l_2 \) in poset \( S \), \( a \) in poset \( S \) is the glb\((l_1, l_2)\) iff
    1) \( a \leq l_1 \) and \( a \leq l_2 \)
    2) for any \( b \) in \( S \), \( b \leq l_1, b \leq l_2 \) implies \( b \leq a \)

- Least upper bound (lub)
  - \( l_1, l_2 \) in poset \( S \), \( c \) in poset \( S \) is the lub\((l_1, l_2)\) iff
    1) \( c \geq l_1 \) and \( c \geq l_2 \)
    2) for any \( d \) in \( S \), \( d \geq l_1, d \geq l_2 \) implies \( d \geq c \)

If glb exists, it is unique. Why? Called meet (denoted by \( \wedge \) or \( \cap \)) of \( l_1 \) and \( l_2 \).

If lub exists, it is unique. Called join (denoted by \( V \) or \( \cup \)) of \( l_1 \) and \( l_2 \).

Definition of a Lattice \((L, \wedge, V)\)

- A lattice \( L \) is a poset under \( \leq \), such that every pair of elements has a glb (meet) and lub (join)

- A lattice need not contain a 0 or 1 element
- A finite lattice must contain 0 and 1 elements
- Not every poset is a lattice
- If there is element \( a \) such that \( a \leq x \) for every \( x \) in \( L \), then \( a \) is the 0 element of \( L \)
- If there is \( a \) such that \( x \leq a \) for every \( x \) in \( L \), then \( a \) is the 1 element of \( L \)

Examples of Lattices

- \( H = (2^D, \cap, U) \) where \( D \) is a finite set
  - glb\((s_1, s_2)\) denoted \( s_1 \cap s_2 \), is set intersection \( s_1 \cap s_2 \)
  - lub\((s_1, s_2)\) denoted \( s_1 \cup s_2 \), is set union \( s_1 \cup s_2 \)
  - \( J = (\mathbb{N}, \gcd, \text{lcm}) \)
    - Partial order is integer divide on \( \mathbb{N} \)
    - lub\((n_1, n_2)\) denoted \( \text{lcm}(n_1, n_2) \)
    - glb\((n_1, n_2)\) denoted \( \gcd(n_1, n_2) \)
      (\( \mathbb{N} \) denotes natural numbers starting at 1)

Chain

- A poset \( C \) where for every pair of elements \( c_1, c_2 \) in \( C \), either \( c_1 \leq c_2 \) or \( c_2 \leq c_1 \)
  - E.g., \( \emptyset \leq \{a\} \leq \{a,b\} \leq \{a,b,c\} \)
  - E.g., from the lattice \( J \) as shown here,
    \[ 1 \leq 2 \leq 6 \leq 30 \]
    \[ 1 \leq 3 \leq 15 \leq 30 \]
  - A lattice s.t. every ascending chain is finite, is said to satisfy the Ascending Chain Condition
Lattices in Dataflow Analysis

- Lattices define property space
- Lattices entail properties of the standard dataflow analysis solution procedure (the worklist algorithm, which we will study shortly).

Dataflow Lattices: Reach

\[ D = \text{all definitions:}\{(x,1),(x,4),(a,3)\} \]

Poset is \( \mathbb{2} \), \( \leq \) is the subset relation

\[
\begin{align*}
1. & x = a \cdot b \\
2. & y \leq a \cdot b \\
3. & a = a + 1 \\
4. & x = a \cdot b \\
5. & \text{goto 3}
\end{align*}
\]

Dataflow Lattices: Avail

\[ D = \text{all expressions:}\{a \cdot b, a + 1, y \cdot z\} \]

Poset is \( \mathbb{2} \), \( \supseteq \) is the superset relation

\[
\begin{align*}
1. & x = a \cdot b \\
2. & y \leq a \cdot b \\
3. & a = a + 1 \\
4. & x = a \cdot b \\
5. & \text{goto 2}
\end{align*}
\]

Dataflow Frameworks

- Equations:
  \[ \text{in}(i) = V \text{out}(i), \quad \text{out}(j) = f_j(\text{in}(j)) \]
  
  where:
  - \( \text{in}(i), \text{out}(j) \) are elements of a property space
  - \( f_j \) is the transfer function associated with node \( j \)
  - \( V \) is the merge operator

Dataflow Frameworks (cont.)

- The property space must be:
  1. A lattice \( L, \leq \)
  2. \( L \) satisfies the Ascending Chain Condition
     Requires that all ascending chains are finite
- The merge operator \( V \) must be the join of \( L \)
- In dataflow, \( L \) is often the lattice of the subsets over a finite set of dataflow facts \( D \)
  - Choose universal set \( D \) (e.g., all definitions)
  - Choose ordering operation \( \leq \). Since the merge operator is must be the join of \( L \), a may problem entails that \( \leq \) is subset. Conversely, a must problem entails that \( \leq \) is superset.

Example: Reach Lattice

- Property space is the lattice of the subsets where
  - \( D \) is the set of all definitions in the program
  - \( \leq \) is the subset operation
    - \( \text{Join} \) is set union \( \cup \), as needed for \( \text{Reach} \), which is a may problem
  - Lattice has \( 0 \) being \( \emptyset \), and \( 1 \) being \( D \)
  - Lattice satisfies the Ascending Chain Condition
**Reach Lattice**

- \( D \) = all definitions: \{(x,1),(x,4),(a,3)\}
- \( \leq \) is the subset relation
- \( 1 \)

**Example: Avail Lattice**

- Property space is the lattice of the subsets where
  - \( D \) is the set of all expressions in the program
  - \( \leq \) is superset
  - \( \text{join} \) of the lattice is set intersection, as needed for \( \text{Avail} \), which is a must problem

- Lattice has \( 0 \) being \( D \), and \( 1 \) being \( \emptyset \)
- Lattice satisfies *Ascending Chain Condition*

**Dataflow Lattices: Avail**

- \( D = \) all expressions: \{a*b,a+1,y*z\}
- \( \leq \) is the superset relation

**Transfer Functions**

- The transfer functions: \( f_j : L \rightarrow L \). Formally, function space \( F \) is such that
  1. \( F \) contains all \( f_j \),
  2. \( F \) contains the identity function \( \text{id}(x) = x \)
  3. \( F \) is closed under composition.
  4. Each \( f_j \) is monotone

**Monotonicity**

- \( F : L \rightarrow L \) is monotone if and only if:
  1. \( a, b \) in \( L \), \( f \) in \( F \) then \( a \leq b \implies f(a) \leq f(b) \)
    or (equivalently):
  2. \( x, y \) in \( L \), \( f \) in \( F \) then \( f(x) \lor f(y) \leq f(x \lor y) \)

- Theorem: Definitions (1) and (2) are equivalent.
  - Show that (1) implies (2)
  - Show that (2) implies (1)

**Distributivity**

- \( F : L \rightarrow L \) is distributive if and only if
  1. \( x, y \) in \( L \), \( f \) in \( F \) then \( f(x \lor y) = f(x) \lor f(y) \)

- Every distributive function is also monotone but not the other way around
  - Distributivity is a very nice property!
Monotonicity and Distributivity

- Is classical Reach distributive?
  - Yes
  - To show distributivity:
    - For each \( j \) \( (\text{in}(j) \cup \text{in}'(j)) \cap \text{pres}(j)) \cup \text{gen}(j) = ((\text{in}(j) \cap \text{pres}(j)) \cup \text{gen}(j)) \cup ((\text{in}'(j) \cap \text{pres}(j)) \cup \text{gen}(j)) \)
    - \( (\text{in}(j) \cup \text{in}'(j)) \cap \text{pres}(j)) \cup \text{gen}(j) = ((\text{in}(j) \cap \text{pres}(j)) \cup \text{gen}(j)) \cup ((\text{in}'(j) \cap \text{pres}(j)) \cup \text{gen}(j)) \)

Spring 19 CSCI 4450/6450, A. Milanova

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if
  - its property space is a lattice \( L, \leq \) that satisfies the Ascending Chain Condition
  - its merge operator \( V \) is the join of \( L \) and
  - its function space \( F: L \rightarrow L \) is monotone
- Thus, we can make use of a generic solution procedure, known as the worklist algorithm or the maximal fixpoint algorithm or the fixpoint iteration algorithm

Worklist Algorithm for Forward Dataflow Problems

/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;
inReach(1) = UNDEF (or {})
for m = 2 to n do
  in(m) = 0
  inReach(m) = {}
W = \{1,2,...,n\} /* put every node on the worklist */
while W \( \neq \) Ø do{
  remove j from W
  out(j) = \( f_j(\text{in}(j)) \)
  for i in successors(j)
    if \( \text{out}(i) \leq \text{in}(j) \) then{
      if \( \text{out}(i) \neq \text{in}(i) \) then{
        \( \text{inReach}(i) = \text{inReach}(i) \cup \text{pres}(i) \cup \text{gen}(i) \)
        \( \text{out}(i) = \text{out}(i) \cup \text{gen}(i) \)
      }
    }
  W = W \cup \{ i \}
}

Worklist Algorithm for Forward Dataflow Problems (slightly different)

/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;
inReach(1) = UNDEF (or {})
for m := 2 to n do
  in(m) = 0
  out(m) = \( f_m(0) \)
W := \{2,...,n\} /* put every node but 1 on the worklist */
while W \( \neq \) Ø do{
  remove j from W
  in(j) = \( \bigvee \{ \text{out}(i) | i \text{ is predecessor of } j \} \)
  out(j) = \( f_j(\text{in}(j)) \)
  if out(j) changed then
    W = W \cup \{ k | k \text{ is successor of } j \}
}

Termination Argument

- Why does the algorithm terminate?
  - Sketch of proof:
    - At each iteration, at least one \( \text{out}(j) \) changes.
    - Since \( \text{out}(j) \) in \( L \), and \( L \) satisfies the Ascending Chain Condition, \( \text{out}(j) \) changes at most \( O(h) \) times where \( h \) is the height of the lattice \( L \)

Correctness Argument

- Theorem: The worklist algorithm computes a solution that satisfies the dataflow equations
  - Why?
    - Sketch of proof:
      - Whenever \( j \) is processed, algorithms sets \( \text{out}(j) = f_j(\text{in}(j)) \). Whenever \( \text{out}(j) \) changes, algorithm puts successors on the list, so \( \text{in}(j) = \bigvee \{ \text{out}(i) \} \).
      - So final solution will satisfy equations.
**Precision Argument**

- **Theorem:** The algorithm computes the least solution of the dataflow equations.
  - Historically though, this solution is often called the maximal fixpoint solution (MFP).
  - I.e., For every node $j$, the worklist algorithm computes a solution $\text{MFP}(j) = \{\text{in}(j), \text{out}(j)\}$, such that every other solution $\{\text{in}'(j), \text{out}'(j)\}$ of the dataflow equations is $\text{in}(j) \leq \text{in}'(j)$, $\text{out}(j) \leq \text{out}'(j)$.

---

**Example**

<table>
<thead>
<tr>
<th>Solution1</th>
<th>Solution2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

1. $z = x + y$
   - $\text{in}_{\text{avail}}(1) = \emptyset$
   - $\text{out}_{\text{avail}}(1) = \{\text{in}_{\text{avail}}(1) \cup \{x+y\}\}

2. if ($z > 500$)
   - $\text{in}_{\text{avail}}(2) = \emptyset$
   - $\text{out}_{\text{avail}}(2) = \emptyset$

3. skip
   - $\text{in}_{\text{avail}}(3) = \emptyset$
   - $\text{out}_{\text{avail}}(3) = \emptyset$

Equivalent to: $\text{in}_{\text{avail}}(2) = \{x+y\}$ and recall that $\cap$ is $\land$ (i.e., set intersection).

---

**Many Applications!**

- Static debugging
  - Memory errors in C/C++ programs
    - Memory leaks
    - Null pointer dereferences
    - Array-out-of-bound accesses
  - Concurrency errors in shared-memory apps
    - Data-races, atomicity violations, deadlocks
- Information flow (as known as taint analysis)

---

**Dataflow Analysis**

- Classical technique
- Compared to Hoare logic, it captures state in a more coarse way
- Still relevant, many interesting problems are phrased in dataflow terms

---

**Next Class**

- MOP vs MFP solutions
- Two classical non-distributive dataflow analyses:
  - Constant propagation and
  - Points-to analysis