Dataflow Analysis: Dataflow Frameworks

Outline of Today’s Class

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

Reading:
- Dragon Book, Chapter 9.2 and 9.3

1. Control-flow graph (CFG):
   - G = (N, E, 1)
   - Nodes are basic blocks
2. Data
3. Dataflow equations
   \[ \text{out}(j) = (\text{in}(j) - \text{kill}(j)) \cup \text{gen}(j) \]
   (gen and kill are parameters)
4. Merge operator V
   \[ \text{in}(j) = V \text{out}(i) \]
   i is predecessor of j

Problem 1: Reaching Definitions

What are the primitive dataflow facts?
Definitions, e.g., \((x, 1), (y, 6)\)
Equations act on sets of definitions.

Problem 2: Live Uses of Variables

What are the primitive dataflow facts?
Variables, e.g., \(x, y, z\).
Equations act on sets of variables.

Problem 3: Available Expressions

What are the primitive dataflow facts?
Expressions, e.g., \(x + y, a * b\).
Equations act on sets of expressions.
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 \( y \).

\[
\begin{array}{c}
\text{n} \\
\text{t1=x op Y} \\
\text{t1=x op Y} \\
\text{t1=x op Y}
\end{array}
\]

Problem 4. Very Busy Expressions (VeryB)

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

\[
\begin{array}{c}
1: x = y + z
\end{array}
\]

Q: What is the data?
Q: What are the equations?
Q: What is \( \text{gen}_{\text{VB}}(j) \)?
Q: What is \( \text{kill}_{\text{VB}}(j) \)?
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 \( j \): \( x = y \text{ op } z \)?
  - \( \text{in}(j) = (\text{out}(j) - \text{kill}(j)) \cup \text{gen}(j) \)
  - \( \text{kill}(j) \): all expressions with operand \( x \): \( (x \text{ op } _), (_\text{ op } x) \)
  - \( \text{gen}(j) \): new expression: \( \{ (y \text{ op } z) \} \)

Very Busy Expressions (VeryB)

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

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 \( n \).
Example: Taint Analysis (explicit flow)

```
x = read();
2. y = 1;
3. x >= 2;
4. y = x * y;
5. x = x - 1;
6. goto 3;
7. z = y - 1;
```

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</strong></td>
<td>Reaching Definitions</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>Problems</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Live Uses of Variables</td>
<td>Very Busy Expressions</td>
</tr>
<tr>
<td>Problems</td>
<td></td>
<td></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:
    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 $D$ (e.g., a definition either reaches $n$ or it does not reach $n$)

Similarities
- Dataflow equations have the same form (from now on, we’ll focus on forward problems):
  - $out(j) = (in(j) \setminus kill(j)) \cup gen(j) = (in(j) \cap pres(j)) \cup gen(j)$
  - $in(j) = \{ V \setminus out(i) \mid i \text{ is predecessor of } j \}$
- $pres(j)$ is the complement of $kill(j)$
  - A note: what makes the 4 classical problems special is that sets $kill(j)/pres(j)$ and $gen(j)$ do not depend on $in(j)$
  - Thus, set union and set intersection can be implemented as logical OR and AND respectively

Similarities
- The dataflow equation at node $j$ is a transfer functions. It takes $in(j)$ as argument and produces $out(j)$ as result:
  - $out(j) = f(in(j))$
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 $\subseteq$)
  - 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

Let $D = \{a, b, c\}$
The poset is $2^D$, $\leq$ is set inclusion

Lattice Theory

- Greatest lower bound (glb)
  - $l_1, l_2$ in poset $S$, $a$ in poset $S$ is the $\text{glb}(l_1, l_2)$ 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 $\text{lub}(l_1, l_2)$ 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 lub exists, it is unique. Called join (denoted by $\lor$ or $\sqcup$) of $l_1$ and $l_2$.

Definition of a Lattice $(L, \Lambda, V)$

- A lattice $L$ is a poset under $\leq$, such that every pair of elements has a $\text{glb}$ (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 $a \leq x$ for every $x$ in $L$, then $a$ is the 0 element of $L$
  - If $x \leq a$ for every $x$ in $L$, then $a$ is the 1 element of $L$

A Poset but Not a Lattice

There is no lub($3, 4$) in this poset so it is not a lattice.
Suppose we add the lub($3, 4$), is it a lattice?
Is This Poset a Lattice

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

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

\[ \begin{array}{c}
\{a\} \\
\{b\} \\
\{c\} \\
\{a, b\} \\
\{a, c\} \\
\{b, c\} \\
\{a, b, c\} \\
\end{array} \]

Examples of Lattices

- \( H = (2^D, \cap, \cup) \) where \( D \) is a finite set
  - \( \text{glb}(s_1, s_2) \) denoted \( s_1 \wedge s_2 \), is set intersection \( s_1 \cap s_2 \)
  - \( \text{lub}(s_1, s_2) \) denoted \( s_1 \vee s_2 \), is set union \( s_1 \cup s_2 \)
- \( J = (\mathbb{N}_1, \text{gcd}, \text{lcm}) \)
  - Partial order is integer divide on \( \mathbb{N}_1 \)
  - \( \text{lub}(n_1, n_2) \) denoted \( n_1 \text{\text{v}}n_2 \) is \( \text{lcm}(n_1, n_2) \)
  - \( \text{glb}(n_1, n_2) \) denoted \( n_1 \Lambda n_2 \) is \( \text{gcd}(n_1, n_2) \)
  (\( \mathbb{N}_1 \) 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., \( \emptyset \leq \{a\} \leq \{a, b\} \)
- 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 \( 2^3 \), \( \leq \) is the subset relation \( \subseteq \)

1. \( x = a \cdot b \)
2. \( 1 \leq y = a \cdot b \)
3. \( a = a + 1 \)
4. \( x = a \cdot b \)
5. goto 3

Dataflow Lattices: Avail

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

Poset is \( 2^3 \), \( \leq \) is the superset relation \( \supseteq \)

1. \( x = a \cdot b \)
2. \( 2 \leq y \cdot z \leq a \cdot b \)
3. \( a = a + 1 \)
4. \( a = a \cdot b \)
5. goto 2
**Dataflow Frameworks**

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

- Other parameters: set of initial CFG nodes, and the values analysis associates with them

---

**Dataflow Frameworks (cont.)**

- **The property space must be:**
  1. A lattice \( L \)
  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 to the join of \( L \), a 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
    - Join is set union \( \cup \), as needed for Reach, which is a *may* problem
  - Lattice has 0 being \( \{\} \), and 1 being \( D \)
  - Lattice satisfies the *Ascending Chain Condition*

**Reach Lattice**

\[
D = \text{all definitions:}\{(x,1),(x,4),(a,3)\} \quad \{x, (x,1),(x,4),(a,3)\} \\
\text{Poset is } P^D, \text{ } \leq \text{ is the subset relation } \subseteq
\]

- 1. \( x=ab \)
- 2. if \( y=a*b \)
- 3. \( a=a+1 \)
- 4. \( x=ab \)
- 5. goto 3

---

**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*
    - Join of the lattice is set intersection, as needed for Avail, which is a *must* problem
  - Lattice has 0 being \( D \), and 1 being \( \{\} \)
  - Lattice satisfies *Ascending Chain Condition*

**Dataflow Lattices: Avail**

\[
D = \text{all expressions:}\{a^*b,a+1,y*z\} \\
\text{Poset is } P^D, \text{ } \leq \text{ is the superset relation } \supseteq
\]

- 1. \( x=a*b \)
- 2. if \( y'=a*b \)
- 3. \( a=a+1 \)
- 4. \( x=a*b \)
- 5. goto 2

---
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 \( a \leq b \iff f(a) \leq f(b) \)
  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!

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if
  1. its property space is a lattice \( L \), \( \leq \) that satisfies the \textit{Ascending Chain Condition}
  2. its merge operator \( \lor \) is the join of \( L \) and
  3. its function space \( F : L \rightarrow L \) is monotone

  Thus, we can make use of a generic solution procedure, known as the \textit{worklist algorithm} or the \textit{maximal fixpoint algorithm} or the \textit{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 \( in(m) = 0 \), \( inReach(m) = \emptyset \);
W = \{1,2,...,n\} /* put every node on the worklist */

while \( W \neq \emptyset \) do {
  remove \( i \) from \( W \)
  \( out(i) = \{ in(i) \} \)
  for \( i \) in successors(\( i \))
    if \( out(j) \nsubseteq \{ in(i) \} \) then {
      \( in(i) = out(j) \lor in(i) \)
      \( W = W \cup \{ i \} \)
    }
  \( outReach(i) = inReach(i) \lor in(i) \)
}
Worklist Algorithm for Forward Dataflow Problems (slightly different)

```c
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;
out(1) = f1(in(1))
for m := 2 to n do
in(m) = 0;
out(m) = fm(0)
W := {2,...,n} /* put every node but 1 on the worklist */
while W ≠ Ø do {
    remove j from W
    in(j) = V {out(i) | i is predecessor of j}
    out(j) = fj(in(j))
    if out(j) changed then
        W = W U { k | k is successor of j }
}
```

Termination Argument

Why does the algorithm terminate?
Sketch of proof:
At each iteration, at least one out(j) changes.
Since out(j) in L, and L satisfies the Ascending Chain Condition, 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 out(j) = f_j(in(j)). Whenever out(j) changes, algorithm puts successors on the list, so in(j) = V {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 MFP(j) = {in'(j), out'(j)}, such that every other solution {in(j),out(j)} of the dataflow equations is in(j) ≤ in'(j), out(j) ≤ out'(j)

Example

```
1. z:=x+y
2. if (z > 500)
3. skip
```

Solution1  Solution2
---  ---
{x+y}  φ
{x+y}  φ
φ  φ

Equivalent to: inavail(2) = {x+y} V inavail(2) and recall that V is ∩ (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)
Many Applications!
- White-box testing: compute coverage
  - Control-flow-based testing
  - Data-flow-based testing
    - Intuitively, test each def-use chain
- Regression testing
  - Analyze changes and select regression tests that actually test changed code

Dataflow Analysis
- Classical and easy to grasp 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