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

Problem 1: Reaching Definitions

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 $n$.

Live Example

What variables are live on exit from statement 3? Statement 1?
Live Uses of Variables (Live)

- Data
  - Primitive facts: variables \( x \)
  - Propagates sets: \( \{x, y, z\} \)
- Dataflow equations. At \( j: x = y+z \)
  - \( \text{kill}_V(j): \{x\} \)
  - \( \text{gen}_V(j): \{y, z\} \)
- Merge operator: set union \( \cup \)

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

\[
\text{in}_V(j) = \left( \text{out}_V(j) - \text{kill}_V(j) \right) \cup \text{gen}_V(j)
\]

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

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

Avail Enables Global Common Subexpressions

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

Data?
- Primitive dataflow facts are expressions, e.g., \(x + y\), \(a \cdot b\), \(a + 2\)
- Analysis propagates sets of expressions, e.g., \(\{x + y, a \cdot b\}\)

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

Example

```
1. \(y = a + b\)
2. \(x = a \cdot b\)
3. if \(y \leq a \cdot b\)
   4. \(a = a + 1\)
   5. \(x = a \cdot b\)
   6. goto 3
7. ...
```

Problem 3: Available Expressions

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

Problem 4: Very Busy Expressions (VeryB)

```
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 \(y\).
```

Very Busy Expressions (VeryB)

```
Data?
- Primitive dataflow facts are expressions, e.g., \(x + y\), \(a \cdot b\)
- Analysis propagates sets of expressions, e.g., \(\{x + y, a \cdot b\}\)

Dataflow equations at \(j\): \(x = y \text{ op } z\)?
- \(\text{in}_{VB}(j) = (\text{out}_{VB}(j) \setminus \text{kill}_{VB}(j)) \cup \text{gen}_{VB}(j)\)
- \(\text{kill}_{VB}(j)\): all expressions with operand \(x\): \((x \text{ op } \_), (\_ \text{ op } x)\)
- \(\text{gen}_{VB}(j)\): new expression: \(\{(y \text{ op } z)\}\)
**Very Busy Expressions (VeryB)**
- Merge operator?
  - For VeryB, it is set intersection $\cap$

  $\text{out}_{\text{VeryB}}(j) = \{ \text{in}_{\text{VeryB}}(i) | i \text{ is a successor of } j \}$

---

**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)**

---

**Outline of Today’s Class**
- Catch up
- Dataflow frameworks
- Lattice
- Transfer functions
- Worklist algorithm
- Reading:
  - Dragon Book, Chapter 9.2 and 9.3

---

**Another Example: Taint Analysis**
- Info leaks

```c
void *fp = &exit;
...
x->f = fp;...
y = x;...
printf("libc function @ %p\n", y->f):
... 0x7f2f8a497030
```
Dataflow Problems

<table>
<thead>
<tr>
<th></th>
<th>May Problems</th>
<th>Must Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Problems</td>
<td>Reaching Definitions</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>Backward Problems</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 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):
  - \( \text{out}(j) = (\text{in}(j) \cap \text{pres}(j)) \cup \text{gen}(j) \)
  - \( \text{in}(j) = \{ \text{V out}(| i \text{ is a predecessor of } j \}) \)
  - \( \text{pres}(j) \) is the complement of \( \text{kill}(j) \) in \( D \)
    - A note: what makes the 4 classical problems special is that sets \( \text{kill}(j) \cap \text{pres}(j) \) and \( \text{gen}(j) \) do not depend on \( \text{in}(j) \)
    - Thus, set intersection and set union can be implemented as logical AND and OR respectively

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 \( b \leq a \) \( \Rightarrow a = b \)
    - Transitive \( a \leq b \) and \( b \leq c \) \( \Rightarrow a \leq c \)
- 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.
**Definition of a Lattice (L, Λ, V)**

A lattice L is a poset under ≤, such that every pair of elements has a \( \text{glb} \) (meet) and \( \text{lub} \) (join).

- A lattice need not contain a 0 or a 1 element
- A finite (height) lattice must contain 0 and 1
- 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 \( x \) 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
  - \( \text{glb}(s1,s2) \) denoted \( s1 \cap s2 \), is set intersection \( s1 \cap s2 \)
  - \( \text{lub}(s1,s2) \) denoted \( s1 \cup s2 \), is set union \( s1 \cup s2 \)
- \( J = (N_1, \gcd, \text{lcm}) \)
  - Partial order is integer divide on \( N_1 \)
  - \( \text{lub}(n1,n2) \) denoted \( n1 \mid n2 \) is \( \text{lcm}(n1,n2) \)
  - \( \text{glb}(n1,n2) \) denoted \( n1 \nmid n2 \) is \( \gcd(n1,n2) \)
- \( N_1 \) denotes natural numbers starting at 1
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., {} $\leq$ \{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
- Lattice properties lead to properties of the standard dataflow analysis solution procedure (the worklist algorithm, which we will study shortly)

Dataflow Lattices: Reach
$D = \text{all definitions:}\{(x,1),(x,4),(a,3)\}$

Position is $2^D$, $\leq$ is the subset relation

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

Dataflow Lattices: Avail
$D = \text{all expressions:}\{a*b,a+1,y*z\}$

Position is $2^D$, $\subseteq$ is the superset relation

1. $x:=a*b$
2. if $y*z <= a*b$
3. $a:=a+1$
4. $x:=a*b$
5. goto 2

Dataflow Frameworks
- Equations:
  \[ \text{in}(j) = V \text{out}(i) \quad \text{out}(j) = f_j(\text{in}(j)) \]
  
  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

- The property space must be:
  1. A lattice $L$, $\leq$
  2. $L$ satisfies the Ascending Chain Condition

- 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 $\subseteq$. Since the merge operator must be the join of $L$, a may problem locks $\subseteq$ to subset. Conversely, a must problem locks $\subseteq$ to superset
**Example: Reach Lattice**

- Property space is the lattice of the subsets where:
  - \( D \) is the set of all definitions in the program
  - \( \subseteq \) 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

---

**Example: Avail Lattice**

- Property space is the lattice of the subsets where:
  - \( D \) is the set of all expressions in the program
  - \( \subseteq \) 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**

- \( D \) = all expressions: \{a*b,a+1,y*z\}
- Poset is \( 2^D \), \( \subseteq \) 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 \( id(x) = x \)
  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 \( a \leq b \) \( \Rightarrow \) \( f(a) \leq f(b) \)
  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

- A function $f: L \to L$ is distributive if and only if $f(x \lor y) = f(x) \lor f(y)$ for all $x, y \in L$.
- 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, $\text{out}(j) = \text{f}(\text{in}(j))$ is $Y = (X \land \text{pres}(j)) \lor \text{gen}(j)$.
  - To show distributivity:
    - For each $j$: $(X \lor (X' \land \text{pres}(j))) \lor \text{gen}(j) = (X \land \text{pres}(j)) \lor \text{gen}(j) \lor (X' \land \text{pres}(j)) \lor \text{gen}(j)$.

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if:
  - Its property space is a lattice $L$ that satisfies the Ascending Chain Condition.
  - Its merge operator $V$ is the join of $L$.
  - Its function space $F: L \to L$ 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;
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 {} do {
    remove j from W
    in(j) = V \{ out(i) | i \in \text{successors}(j) \}
    out(j) = \text{f}_j(\text{in}(j))
    if \text{out}(j) \neq \text{in}(j) then {
        in(j) = \text{out}(j) \lor \text{v}(\text{in}(j))
        W = W \cup \{ i \}
    }
}
```

Termination Argument

- Why does the algorithm terminate?
- Sketch of proof:
  - At each iteration, no $\text{in}(j)$ nor $\text{out}(j)$ shrinks, and at least one $\text{out}(j)$ grows.
  - Since $\text{out}(j)$ in $L$, and $L$ satisfies the Ascending Chain Condition, $\text{out}(j)$ changes at most $O(\text{h})$ times where $\text{h}$ 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, algorithm 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>( \text{in}_{\text{avail}}(1) = \emptyset )</th>
<th>Solution 1</th>
<th>Solution 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{out}<em>{\text{avail}}(1) = (\text{in}</em>{\text{avail}}(1) \cup { x+y }) )</td>
<td>( { x+y } )</td>
<td>( { x+y } )</td>
</tr>
<tr>
<td>( \text{in}<em>{\text{avail}}(2) = \text{out}</em>{\text{avail}}(1) \lor \text{out}_{\text{avail}}(3) )</td>
<td>( { x+y } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{out}<em>{\text{avail}}(2) = \text{in}</em>{\text{avail}}(2) )</td>
<td>( { x+y } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{in}<em>{\text{avail}}(3) = \text{out}</em>{\text{avail}}(2) )</td>
<td>( { x+y } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{out}<em>{\text{avail}}(3) = \text{in}</em>{\text{avail}}(3) )</td>
<td>( { x+y } )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Equivalent to: \( \text{in}_{\text{avail}}(2) = \{ x+y \} \lor \text{in}_{\text{avail}}(2) \) and recall that \( \lor \) (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 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