Data-Flow Analysis II
Data-Flow Analysis Review

**Goal:** Model program state along all program paths.

**Concern:** *Undecidable*. Also, number of paths is exponential.

**Approach:**
- Consider subset of state (*data-flow value*).
- Reduce paths: $\text{IN}[b] = \wedge_{a \text{ precedes } b} \text{OUT}[a]$ (*meet operator*).
- Compute: $\text{OUT}[b] = f_b(\text{IN}[b])$ (*transfer function*).
- Necessarily approximate solution.
# The Meet Operator and Its Domain

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idempotent</td>
<td>$x \land x = x$</td>
</tr>
<tr>
<td>Commutative</td>
<td>$x \land y = y \land x$</td>
</tr>
<tr>
<td>Associative</td>
<td>$x \land (y \land z) = (x \land y) \land z$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top (T)</td>
<td>$\forall x. T \land x = x$</td>
</tr>
<tr>
<td>Bottom (⊥)</td>
<td>$\forall x. \bot \land x = \bot$</td>
</tr>
</tbody>
</table>
The Meet Operator and Its Domain

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idempotent</td>
<td>$x \land x = x$</td>
</tr>
<tr>
<td>Commutative</td>
<td>$x \land y = y \land x$</td>
</tr>
<tr>
<td>Associative</td>
<td>$x \land (y \land z) = (x \land y) \land z$</td>
</tr>
<tr>
<td>Element</td>
<td>Top ($\top$)</td>
</tr>
<tr>
<td></td>
<td>$\forall x . \top \land x = x$</td>
</tr>
<tr>
<td></td>
<td>Bottom ($\bot$)</td>
</tr>
<tr>
<td></td>
<td>$\forall x . \bot \land x = \bot$</td>
</tr>
</tbody>
</table>
Meet Semilattices

We can define a partial order:
- Reflexive, antisymmetric, transitive.
- \( x \leq y \equiv x \land y = x \)

Greatest Lower Bound (glb)
- \( \text{glb}(x, y) = x \land y \)

\[ T = \{a,b,c\} \]
\[ \{a,b\}, \{a,c\}, \{b,c\} \]
\[ \{a\}, \{b\}, \{c\} \]
\[ \bot = \{\} \]
\[ x \subseteq y \equiv x \cap y = x \]
## Transfer Functions

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity Function</td>
<td>$\exists I \in F. \forall x \in V. I(x) = x$</td>
</tr>
<tr>
<td>Closed under Composition</td>
<td>$\forall f, g \in F. h(x) = g(f(x)) \Rightarrow h \in F$</td>
</tr>
<tr>
<td>Monotone (1)</td>
<td>$\forall x, y \in V. \forall f \in F$</td>
</tr>
<tr>
<td></td>
<td>$f(x \land y) \leq f(x) \land f(y)$</td>
</tr>
<tr>
<td>Monotone (2)</td>
<td>$\forall x, y \in V. \forall f \in F$</td>
</tr>
<tr>
<td></td>
<td>$x \leq y \Rightarrow f(x) \leq f(y)$</td>
</tr>
</tbody>
</table>
### Transfer Functions

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity Function</td>
<td>$\exists I \in F. \forall x \in V. I(x) = x$</td>
</tr>
<tr>
<td>Closed under Composition</td>
<td>$\forall f, g \in F. h(x) = g(f(x)) \Rightarrow h \in F$</td>
</tr>
<tr>
<td>Monotone (1)</td>
<td>$\forall x, y \in V. \forall f \in F$</td>
</tr>
<tr>
<td></td>
<td>$f(x \land y) \leq f(x) \land f(y)$</td>
</tr>
<tr>
<td>Monotone (2)</td>
<td>$\forall x, y \in V. \forall f \in F$</td>
</tr>
<tr>
<td></td>
<td>$x \leq y \Rightarrow f(x) \leq f(y)$</td>
</tr>
</tbody>
</table>

Needed for termination
Statements vs. Basic Blocks

We often define transfer functions for *statements* instead of *basic blocks*.

- If basic block $B = \langle s_1, s_2, ... s_n \rangle$, then $f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1}$

Data-flow analysis does not require *maximal* blocks.
- Same result if each block is one statement.

Basic blocks are an *optimization*: fewer nodes in the graph.
Forward Data-Flow Algorithm

Given:
- \(V\): values of lattice
- \(\wedge\): meet operator
- \(F\): set of transfer functions
- CFG with unique Entry and Exit nodes
- \(v_{\text{ENTRY}}\): data-flow value for Entry node

For each block \(b\), \(\text{OUT}[b] = T\)
\(\text{OUT}[\text{ENTRY}] = v_{\text{ENTRY}}\)

While any \(\text{OUT}\) changes

For each block \(b\) except Entry

\(\text{IN}[b] = \wedge_a \text{OUT}[a]\)
\(\text{OUT}[b] = f_b(\text{IN}[b])\)
Backward Data-Flow Algorithm

Given:
- $V$: values of lattice
- $\wedge$: meet operator
- $F$: set of transfer functions
- CFG with unique ENTRY and EXIT nodes
- $v_{\text{EXIT}}$: data-flow value for EXIT node

For each block $b$, $\text{IN}[b] = T$
$\text{IN}[\text{EXIT}] = v_{\text{EXIT}}$

While any $\text{IN}$ changes
  For each block $b$ except EXIT
    $\text{OUT}[b] = \bigwedge_c \text{IN}[c]$
    $\text{IN}[b] = f_b(\text{OUT}[b])$
Live Variable Analysis

Goal: Determine range of statements in which a value may be needed.

Used in:
- Dead code elimination.
- Register allocation.
Live Variable Analysis

Direction: Backward

Values: Set of live locations.
- \( V \subseteq \{ r_i | 0 \leq i \leq 7 \} \cup \{ sp[j] | 0 \leq j \} \)

Meet operator: set union

Transfer functions:
- \( op \ ra < - rb \ rc \)
- \( f(x) = \{ rb, rc \} \cup (x - \{ ra \}) \)
Constant Propagation

Direction: Forward

Values:
- $\langle r0, r1, \ldots, r7, sp[j], \ldots \rangle$
- $\nu_i \in \{\text{T(unknown)}, \perp(\text{nac})\} \cup \mathbb{Z}$

Meet operator:
- $\langle \ldots, x_i, \ldots \rangle \land \langle \ldots, y_i, \ldots \rangle =$
  - Usual rules for $\text{T}$ and $\perp$
  - $c$ if $x_i = y_i = c$
  - $\perp$ otherwise
Constant Propagation

Transfer Functions:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>li ri &lt;- c</td>
<td>?</td>
</tr>
<tr>
<td>ld ri &lt;- sp[j]</td>
<td>?</td>
</tr>
<tr>
<td>st sp[i] &lt;- rj</td>
<td>?</td>
</tr>
<tr>
<td>mul ra &lt;- rb rc</td>
<td>?</td>
</tr>
<tr>
<td>call ri</td>
<td>?</td>
</tr>
</tbody>
</table>
Redundant Expressions

Global Common Expressions

mul r1 <- r2 r3
jmp L2

L1:
  mul r4 <- r2 r3

L2:
  mul r5 <- r2 r3
Redundant Expressions

Global Common Expressions

mul r1 <- r2 r3
jmp L2

L1:
mul r4 <- r2 r3

L2:
mul r5 <- r2 r3

r2*r3 is already computed on every path
Redundant Expressions

Global Common Expressions

```
mul r1 <- r2 r3  
jmp L2

L1:  
mul r4 <- r2 r3

mul r6 <- r2 r3  
mov r1 <- r6  
jmp L2

mul r6 <- r2 r3  
mov r4 <- r6

L2:  
mul r5 <- r2 r3

L2:  
mov r5 <- r6
```
Redundant Expressions

Global Common Expressions

```
L2: mul r5 <- r2 r3
    mul r6 <- r2 r3
    mov r5 <- r6
    jmp L2
L1: mul r4 <- r2 r3
    mov r1 <- r6
    jmp L2
```

Clean up with copy propagation.
Redundant Expressions

Loop Invariant Expressions

L1:
bz r0 L2

... div r1 <- r2 r3 ...
jmp L1

r2 and r3 are not modified in loop
Redundant Expressions
Loop Invariant Expressions

L1:
bz r0 L2
...
div r1 <- r2 r3
... jmp L1
...

L1:
bz r0 L2
...
mov r1 <- r4
... jmp L1
...

div r4 <- r2 r3
Redundant Expressions

Loop Invariant Expressions

L1:
bz r0 L2

... div r1 <- r2 r3 ...
... jmp L1

... div r4 <- r2 r3

... mov r1 <- r4 ...
... jmp L1

Could cause divide-by-zero!
Redundant Expressions

Loop Invariant Expressions

L1:
  bz r0 L2

... div r1 < - r2 r3
... jmp L1

... bz r0 L2
div r4 < - r2 r3

L1:
... mov r1 < - r4
... bnz r0 L1

...
Redundant Expressions

Partially Redundant Expressions

add r1 <- r2 r3

... Assume r2 and r3 are not modified.

add r4 <- r2 r3

...
Redundant Expressions

Partially Redundant Expressions

```
add r1 <- r2 r3
...
add r4 <- r2 r3
...
```

Move expression here?
Redundant Expressions

Partially Redundant Expressions

```
add r1 <- r2 r3
...  ...  add r5 <- r2 r3
...  mov r1 <- r5
...  mov r4 <- r5
...  add r4 <- r2 r3
...  ...  ...  ...
```
Code Motion and Debugging

We are changing the order of evaluation.

- “Don’t break the build” – all valid runs must still be valid.
- Evaluate expressions only if the naïve code would.

What about reordering invalid runs?

- E.g., an exception gets moved after database update.
- Need to maintain sequence of user-visible state changes.

This is why debugging optimized code is not always obvious.
Lazy Code Motion

1. Find anticipated expressions at each program point $p$.
   - I.e., all $e$ such that all paths from $p$ eventually compute $e$.

2. Determine available expressions at each point $p$.

3. Postpone expressions as long as possible.

4. Eliminate unused temporaries.
**Anticipated Expressions**

**Direction:** Backward

**Values:** Sets of expressions

**Meet operator:** \( \cap \)

\( v_{\text{EXIT}} = \{ \} \)

**Transfer function:**

\( f_b(x) = use_b \cup (x - kill_b) \)

**Use set:**

\( use_b = \{ e | e \text{ is computed in } b \} \)

**Kill set:**

\( kill_b = \{ e | \exists x. \text{isop}(x, e) \land \text{def}(x, b) \} \)
Available Expressions

Direction: Forward

Values: Sets of expressions

Meet operator: $\cap$

$v_{\text{ENTRY}} = \{\}$

Transfer function:
- $f_b(x) = available[b] − kill_b$

After this analysis, insert expressions at points where the expression is first anticipated.
Postponable Expressions

Direction: Forward

Values: Sets of expressions

Meet operator: $\cap$

$v_{ENTRY} = {}$

Transfer function:

$\odot f_b = (earliest[b] \cup x) - use_b$

$earliest[b] = anticipated[b] - available[b]$
Used Expressions
Direction: Backward
Values: Sets of expressions
Meet operator: $\cup$
$v_{\text{EXIT}} = \{\}
Transfer function:
$\circ f_b(x) = (use_b \cup x) - latest[b]$