Iterative Data Flow Analysis

- Start with initial guess of info at loop head
  \[ \text{info}_{\text{loop-head}} = \text{guess} \]
- Solve equations for body
  \[ \text{info}_{\text{loop-head}} = F_{\text{body}}(\text{info}_{\text{loop-head}}) \]
  \[ \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}} \]
- Test if fixed-point found
  \[ \text{info}_{\text{loop-head}}' = \text{info}_{\text{loop-head}} \]
  - If same done
  - Else use result as new (better) guess:
    \[ \text{info}_{\text{back-edge}}' = F_{\text{body}}(\text{info}_{\text{loop-head}}) \]
    \[ \text{info}_{\text{loop-head}}'' = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}} \]

When does Iterating work?

- Have to be able to make initial guess
- Info^{n+1} must be closer to fixed-point than info^n
- Must eventually reach fixed-point, info must be drawn from a finite height domain

To reach best fixed-point, initial guess should be optimistic:
- \[ \text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \]
- Even if guess is overly optimistic, iteration will ensure that we get a safe
- Iteration speed
  - Ideal: solve constraints along shortest path from loop head to tail
  - Practical: avoid solving constraints outside of loop until fixed point reached inside loop
Data Flow Analysis Direction

- Constraints are declarative, so may require mix of forward and backward
- Frequently directional propagation & iteration
  - Forward or backward
  - Topological traversal of acyclic subgraph
    minimizes analysis time
- Directional constraints are called flow functions
  \[ \text{RDEF}_{s\leftarrow s'}(\text{in}) = \text{in} - (s\leftarrow s') \forall x' \cup (s\leftarrow s) \]

GEN & KILL sets

- Flow functions can often be described by their GEN and KILL sets
  - GEN = new information added
  - KILL = old information removed
  - \( F_{\text{in}}(\text{in}) = \text{in} - \text{KILL}_{\text{in}} \cup \text{GEN}_{\text{in}} \)
  - Example reaching definitions
    \[ \text{GEN}_{x\leftarrow s} = \{x\leftarrow s\} \]
    \[ \text{KILL}_{x\leftarrow s} = \{x\leftarrow s'| \forall x'\} \]

Bit Vectors

- Encode data flow information and GEN/KILL sets as bit vectors
  - Works when info can be expressed abstractly as a set of things
  - Each gets a specific bit position
  - Reaching defs: info = bit vector over statements, each bit represents a specific statement, defined variable is implied by statement
- Advantages
  - Compact representation
  - Fast union & difference operations
  - Can combine GEN / KILL sets of whole basic block into one GEN/KILL set for faster iteration
Example: Constant Propagation

- What info should be represented?
- \( CP_{x:=y*5}(n) = \)
- Merge function?
- Initial info?
- Direction?
- Can bit vectors be used?

Another Example: Constant Propagation

<table>
<thead>
<tr>
<th>x := 3</th>
<th>w := 3</th>
<th>y := x*2</th>
<th>z := y+5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := x+1</td>
<td>w := 3</td>
<td>y := x*2</td>
<td>z := y+5</td>
</tr>
<tr>
<td>w := w*2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

May vs. Must Info

- Some kind of info implies guarantees: **must info**
- Some kind of info implies possibilities: **may info**

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>desired info</td>
<td>small set</td>
<td>big set</td>
</tr>
<tr>
<td>safe</td>
<td>overly large set</td>
<td>overly small set</td>
</tr>
<tr>
<td>GEN</td>
<td>add everything that might be true</td>
<td>add only if guaranteed true</td>
</tr>
<tr>
<td>Kill</td>
<td>remove only if guaranteed wrong</td>
<td>remove everything possibly wrong</td>
</tr>
<tr>
<td>Merge</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Live Variables Analysis

- Desired info: set of variables *live* at each program point
  - Live = might be used later in the program
  - Supports dead assignment elimination, register allocation
- May or must?
- Direction?
- Merge function
- Bit vectors usable?
- Initial info?

Live Variables Example

```
x := 5
y := x*2
x := x+1
y := x+10
```

Lattice-theoretic Data Flow Analysis Framework

- **Goals**
  - Provide single, formal model to describe all DFAs
  - Formalize "safe", "conservative", "optimistic"
  - Precise bounds on time complexity
  - Connect analysis with underlying semantics to enable correctness proofs
- **Plan**
  - Define domain of program properties computed by DFA
  - Each domain has set of elements
  - Each element represents one possible value of the property
  - Order sets to reflect relative precision
  - Domain = set of elements + order over elements = lattice
  - Define flow/merge functions using lattice operators
  - Benefit from lattice theory for realizing goals
Lattices

- $D = (S, \leq)$
  - $S$ set of elements
  - $\leq$ induces a partial order
    - Reflexive, transitive & anti-symmetric
    - $\forall x, y \in S: \text{meet}(x, y)$ (greatest lower bound)
    - $\text{join}(x, y)$ (least upper bound)
- Unique Top (T) & Bottom ($\bot$) elements:
  - $X \bot = \bot$ and $x \vee T = T$
  - Meet and join are commutative and associative
- Height of lattice: longest path through partial order from top to bottom

Lattices in Data Flow Analysis

- Model information by elements of a lattice domain
  - Top = best case info
  - Bottom = worst case info
  - Initial info for optimistic analyses (at least back edges: top)
  - If $a \leq b$ then $a$ is a conservative approximation of $b$
  - Merge function $= \text{meet}(\cdot)$: the most precise element that’s a conservative approximation of both input elements
- Initial info for optimistic analyses (at least back edges: top)

Some Typical Lattice Domains

- Two point lattice: $\bot, T$
  - Boolean property
  - A tuple of two point lattices = bit vector
  - Lifted set: set of incomparable values and $\bot$ and $T$
  - Example?
  - Powerset lattice: set of all subsets of $S$, ordered somehow (often by $\subseteq$)
    - $T = \emptyset \bot = S$ or vice versa
  - Collecting analysis
    - Isomorphic to tuple of booleans indicating membership in subset of elements of $S$
Product (aka. Tuple) Lattices

- Often useful to break complex lattice into a tuple of lattices, one per variable analyzed.
- \( D_T = <S_1, S_2> = <S_1, S_2>^N \)
  - \( S_i \) = \( S_1 \times S_2 \times \ldots \times S_N \)
  - \( \leq \) pointwise ordering
- \( T_T = <T_1, \ldots, T_N> \), bottom tuple of bottoms
- \( \text{Height}(D_T) = N \times \text{height}(D) \)
- Example?

Analysis of Loops with Lattices

- Flow function for loop body 
- \( F(\text{info-at-loop-head}) = \text{info at back edge:} \)
- \( F(0) = d_{\text{entry}} \wedge T \)
- \( F(1) = d_{\text{entry}} \wedge \text{back}(F(0)) \wedge F(\text{entry}) = F(d_{\text{entry}}) \)
- \( F(k) = d_{\text{entry}} \wedge \text{back}(F(k-1)) = F(F(\ldots(\text{entry})\ldots)) \)
- Repeat until \( F(k+1) = F(k) \)

Termination of Iterative Analysis

- Sufficient conditions
  - Flow functions \( F \) are monotonic 
    - \( d_1 \preceq d_2 \Rightarrow F(d_1) \preceq F(d_2) \)
  - Lattice is of finite height 
    - Start at \( T \)
    - Each application of \( F \) goes down one level
    - Eventually hit fixed-point or bottom
      - At most \( \text{height}(D) - 1 \) applications
Examples

- Lattices for
  - Constant propagation
  - Live variables
  - Reaching definitions

Distributive Lattices

- A lattice is **distributive**: $\forall x, y, z \in D: (x \land y) \lor z = (x \lor z) \land (y \lor z)$
  and $(x \lor y) \land z = (x \land z) \lor (y \land z)$

  - Example:
    - Live variables, only elements $T$ and $\bot$ (easy exercise)
  - Counterexample
    - Constant propagation:
      - $(1 \lor 2) \land 3 = (1 \land 3) \lor (2 \land 3)$

Meet-Over-All Paths Solution

- Flow function for basic block $B$: $F_{B_p}$ for a path $p$ along
  $B_1 \ldots B_n$: $F = F_{B_1} \ldots F_{B_n}$

  - $\text{MOP}(B) = \bigwedge_{p \in \text{paths}(B)} F_p(\text{Init})$
    Init is initial info at entry block
  - $\text{MOP}$ is most precise solution we can hope for
  - $\text{MOP}$ computation is NP for monotone flow functions
  - I.e. there is no algorithm that is guaranteed to work for all flow graphs
  - Use approximation: **maximum Fixed-Point solution** ($\text{MFP}$)
### Important Results

- **Monotone lattices**
  - Iterative algorithm guaranteed to produce the MFP solution

- **Distributive monotone lattices**
  - MFP = MOP
  - Functions over lattices of bit vectors are distributive, i.e., all functions \( f: \text{BV}^n \rightarrow \text{BV}^m \) are distributive

### Important Data Flow Problems

- **Reaching definitions**
  - Which definitions of a variable \( v \) reach a particular use of \( v \)

- **Available expressions**
  - What expressions are available at a particular program point (e.g., \( x^y \) is available in variable \( t_1 \))

- **Live variables**
  - For a given program point, is there a use of the variable along some path to exit

- **Upwards exposed uses**
  - What uses of variables at particular points are reached by particular definitions

- **Copy propagation**
  - For \( x := y \) to a use of \( x \) no assignments to \( y \)?

- **Constant propagation**

- **Partial redundancy elimination**
  - Original formulation forward & backward flow problem

### Worklist Algorithm for IDFA

```plaintext
procedure Worklist_Iterate(N, Entry, FP, dfin, Init)
N: in set of Node
entry: in Node
FP: in Node -> L
dfin: out Node -> L
Init: in L
begin
B, P: Node
Worklist: set of Node
effect, totaleffect: L
dfin(entry) := Init
Worklist := N - {entry}
for each \( B \) in \( N \) do
dfin(\( B \)) := TOP
od
repeat
B := choose(Worklist)
Worklist := Worklist - {B}
totaleffect := r
for each \( P \) in Pred(\( B \)) do
effect := F(\( P \), dfin(\( P \)))
totaleffect := MEET\( = \) effect
if dfin(\( B \)) \#= totaleffect then
dfin(\( B \)) := totaleffect
Worklist := Worklist \cup \{B\}
fi
od
until Worklist = {}
end
```
Lattices of Flow Functions (1)

- Can define lattice of monotone flow functions over lattices:
  - L lattice, define L set of monotone functions from L to L, i.e., f : L → L ⇒ ∀x,y ∈ L x ≤ y ⇒ f(x) ≤ f(y)
  - Meet defined as: ∀ f,g ∈ L, ∀ x ∈ L:
    \( (f \wedge g)(x) = f(x) \wedge g(x) \)
  - Top: ∀ x ∈ L: T(x) = T
  - Bottom: ∀ x ∈ L: \( \bot(x) = \bot \)

Lattices of Flow Functions (2)

- Identify function: id(x) = x, ∀ x ∈ L as
- Function composition: (f o g)(x) = f(g(x))
  - L is closed under composition
  - f^0 := id, f^n := f o f^{n-1} for n ≥ 1
- Kleene closure f^*
  - ∀ x ∈ L: f^*(x) := lim_{n→∞} (id^o f)^n(x)
  - L is closed under Kleene closure (for finite height lattices)
    - Sufficient to have finite effective height (relative to function f)
    - longest strictly descending chain of the form f(x), f^2(x), ...

Control-Tree-Based Data Flow Analysis

- Recall two approaches for control flow analysis
  - Interval analysis
  - Structural analysis
- Control-tree-based data flow analysis uses the intervals / control structures identified to perform data flow analysis
Structural Data Flow Analysis

- Example: if-then

- $F_{\text{if-then}} = (F_{\text{then}} \circ F_{\text{IF}}) \land F_{\text{FN}}$

- Propagate info to substructures:
  - $\text{in}(\text{if}) = \text{in}(\text{if-then})$
  - $\text{in}(\text{then}) = F_{\text{IF}}(\text{in}(\text{if}))$

While Loops

- $F_{\text{while-loop}} = F_{\text{while}} \circ (F_{\text{loop}} \circ F_{\text{while-loop}}) \land F_{\text{loop}}$

- Propagate info to substructures:
  - $\text{in}(\text{while}) = F_{\text{loop}}(\text{in}(\text{while-loop}))$
  - $\text{in}(\text{body}) = F_{\text{while-loop}}(\text{in}(\text{while}))$

Structural Analysis for Backward Flow Problems

- Problem
  - Single-entry but multiple-exit control structures
    - Have multiple entries in backward flow
    - Have to meet (\^) possible exits
      - Details in the book
  - Single-entry, single-exit structures
    - Can turn equations around