Lattice-Theoretic Data Flow Analysis Framework

Basic idea:

- define **domain** of program properties computed by DFA
- organize elements into **lattice**
- define flow functions, merge function over this domain

Benefits:

+ get nice guarantees about iterative analysis convergence
+ get nice semantic reasoning properties
+ produce framework that unifies all data flow analyses

Lattices

Define lattice $D = (S, \le)$:

- S is a set of elements of the lattice
- $\le$ is a binary relation over elements of S

Properties of $\le$:

- $\le$ induces a partial order over S
  - reflexive, transitive, & anti-symmetric
- every pair of elements of S has
  a unique greatest-lower-bound (aka meet) and
  a unique least-upper-bound (aka join)

Height of D =
longest path through partial order from greatest to least

- infinite lattice can have finite height (but infinite width)

Top (T) = unique element of S that's greatest, if exists
Bottom ($\perp$) = unique element of S that's least, if exists

Lattice models in data flow analysis

Model information by elements of a lattice

- top = best case info
- bottom = worst case info
- merge function = glb (meet) on lattice elements
- if $a \le b$, then a is a conservative approximation to b

Reaching definitions:

- elements:
- meet:
- top:
- bottom:
- $\le$:

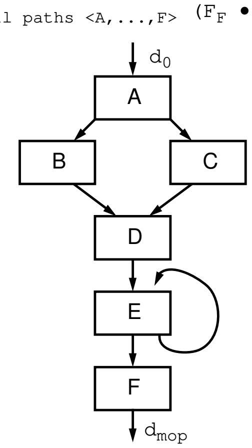
Reaching constants:

- elements:
- meet:
- top:
- bottom:
- $\le$:

Solving data flow analyses

Ideally, want **meet-over-all-paths** at each program point:

$$d_{mop} = \bigcap_{\text{all paths } \langle A,\ldots,F\rangle} (F_F \bullet \ldots \bullet F_A)(d_0)$$

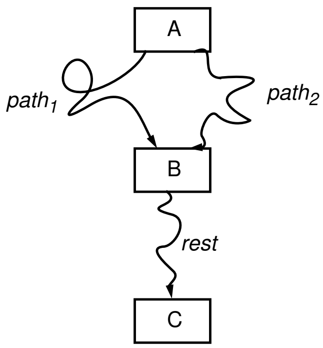


Unfortunately, MOP is expensive to compute

- merges lead to reanalyzing stmts after merge for each
  predecessor path $\Rightarrow$ exponential blow-up
- loops generate infinite # of paths

Approximation for merges

Compute meets early (at merge points) rather than at end



$d_{mop} = F_{rest}(d_{path1}) \cap F_{rest}(d_{path2})$

$d_{approx} = F_{rest}(d_{path1} \cap d_{path2})$

Questions:
- is this approximation safe?
- is this accurate?
- is this efficient?

Legality & accuracy of early merging

Is d_{approx} legal?

I.e., is $d_{approx} \le d_{mop}$?

True if:

$F_{rest}(d_{path1} \cap d_{path2}) \le F_{rest}(d_{path1}) \cap F_{rest}(d_{path2})$

By definition, this is true if F_{rest} is **monotonic**

If F_{rest} is **distributive**, i.e.

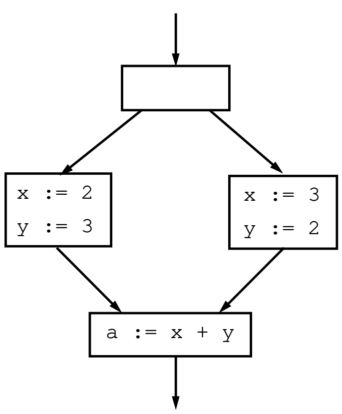
$F_{rest}(d_{path1} \cap d_{path2}) = F_{rest}(d_{path1}) \cap F_{rest}(d_{path2})$

then approximation loses no information

Reaching definitions?

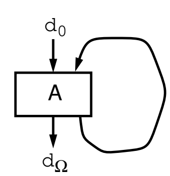
Reaching constants?

Example



Approximation for loops

Use iteration to unroll a fixed number of paths through loop, till fixed point is reached in analysis



ideal MOP solution:

$$d_{\Omega\text{-}mop} = F_A(d_0) \cap F_A{\cdot}F_A(d_0) \cap F_A{\cdot}F_A{\cdot}F_A(d_0) \cap \ldots \cap F_A^{\,k}(d_0) \cap \ldots \cap F_A^{\,\infty}(d_0)$$

approximation:

$$d_{\Omega\text{-}approx} = F_A(d_0) \cap F_A{\cdot}F_A(d_0) \cap \ldots \cap F_A^{\,k}(d_0)$$

where $F_A^{\,k}(d_0) = F_A^{\,k+1}(d_0)$

Questions:
- is this approximation safe?
- is this accurate?
- is this efficient?

Legality & accuracy of iterative approximation

$d_{\Omega\text{-approx}} \le d_{\Omega\text{-mop}}$ if

$F_A{}^k(d_0) \le F_A{}^k(d_0) \cap \ldots \cap F_A{}^\infty(d_0)$,

where $F_A{}^k(d_0) = F_A{}^{k+1}(d_0)$

True if F_A **idempotent** and finite k exists

Will iteration finish?

Is k finite? Is k small?

Lattice gives answer:
$k \le \text{height}(D)$

If lattice *D* of finite height,
and flow functions are monotonic and idempotent, then
iteration is guaranteed to terminate with safe answer

Complexity of iterative analysis

How long does iterative analysis take?

n: # of stmts in loop
t: time to execute one flow function
l: depth of loop nesting
k: height of lattice

Another example: integer range analysis

For each program point,
for each integer-typed variable,
calculate (an approximation to) the set of integer values
that can be taken on by the variable

Use info for constant folding comparisons,
for eliminating array bounds checks,
for (in)dependence testing of array accesses,
for eliminating overflow checks

Example

```
for i := 0 to N-1
  ... a[i] ...
end
```