Lattice-Theoretic Data Flow Analysis Framework

Basic idea:

- define **domain** of program properties computed by DFA
  - elements of domain organized 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

Preorders, partial orders, and lattices

Preorder: binary relation that's **reflexive** and **transitive**

Partial order: preorder that's **antisymmetric**

- no cycles in relation

Lattice: partial order where every pair of elements has
a **greatest lower bound** (glb, meet) and
a **least upper bound** (lub, join)

- top, T = lub(all elements)
- bottom, $\perp$ = glb(all elements)

Downward (upward) **semi-lattice**: p.o. with
greatest lower bounds (least upper bounds, resp.)

Any of these can be infinite

- height of infinite p.o. can be infinite or finite

Lattice models in data flow analysis

Model information by elements of a lattice

Convention (opposite to denotational semantics!):

- merge function = meet on lattice elements
- top = best case info
- bottom = worst case info

Reaching definitions:

- lattice domain:
- meet:
- top:
- bottom:
- ≤:

Reaching constants:

- lattice domain:
- meet:
- top:
- bottom:
- ≤:

Constructing data flow analyses

Define each flow function over lattice domain `D`

`F: RTL → D → D`

Define initial domain element d_0 at start of CFG

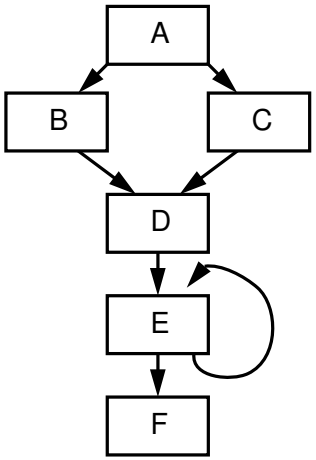
E.g. for reaching constants

```
F "var := const" d =
  d - (var → *) ∪ {(var → const)}
F "var := var2 op var3" d = ...
...
```

$d_0 = \emptyset$

Solving data flow analyses

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



Unfortunately, MOP is expensive to compute

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

Approximation for merges

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

Questions:

- is this legal (sound, conservative)?
- is this accurate?
- is this efficient?

Legality & accuracy of early merging

Is d_{approx} legal?

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

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

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

$$d_{approx} \leq d_{mop} \Leftarrow$$

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

i.e. F_{rest} (flow functions) are **monotonic**

Moreover, if F_{rest} is **distributive**, i.e.

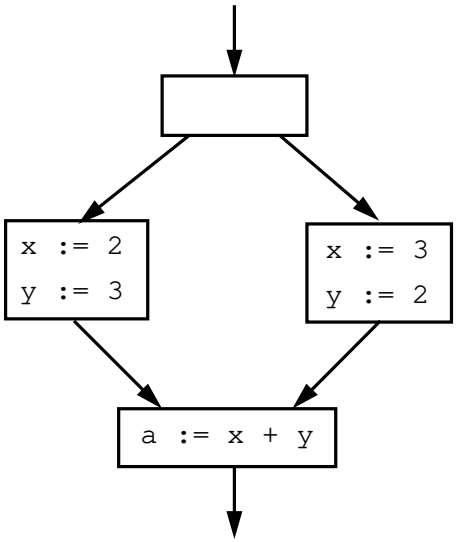
$$F_{rest}(d_{path1} \cap d_{path2}) = F_{rest}(d_{path1}) \cap F_{rest}(d_{path2})$$

then approximation is just as accurate as mop solution

Reaching definitions?

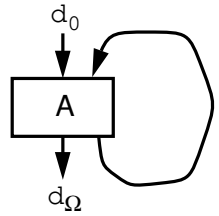
Reaching constants?

Example



Approximation for loops

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



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 legal (sound, conservative)?
- is this accurate?
- is this efficient?

Legality & accuracy of iterating

$$d_{\Omega\text{-approx}} \leq d_{\Omega\text{-mop}} \quad \Leftarrow$$

$$F_A^k(d_0) \leq F_A^k(d_0) \cap \ldots \cap F_A^\infty(d_0)$$

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

$$F_A^k(d_0) = F_A^k(d_0) \cap \ldots \cap F_A^\infty(d_0)$$

true, if F_A idempotent and finite k exists

Will iteration finish?

Is `k` finite? Is `k` small?

Lattice gives answer:

`k ≤ height(D)`

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

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

Ideas:

- maintain set of values for each variable
- maintain [low..high] range for each variable