Lattice-Theoretic Data Flow Analysis Framework

Goals:

- provide a single, formal model that describes all DFAs
- formalize notions of “safe”, “conservative”, “optimistic”
- place precise bounds on time complexity of DF analysis
- enable connecting analysis to underlying semantics for correctness proofs

Plan:

- define **domain** of program properties computed by DFA
  - domain has a set of elements
  - each element represents one possible value of the property
  - (partially) order elements to reflect their relative precision
  - domain = set of elements + order over elements = **lattice**
- define flow functions & merge function over this domain, using standard lattice operators
- benefit from lattice theory in attacking above issues

History: Kildall [POPL 73], Kam & Ullman [JACM 76]

Lattices

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

- S is a (possibly infinite) set of elements
- $\leq$ is a binary relation over elements of S

Required properties of $\leq$:

- $\leq$ is a **partial order**
  - reflexive, transitive, & anti-symmetric
- every pair of elements of S has
  a unique **greatest lower bound** (a.k.a. meet) and
  a unique **least upper bound** (a.k.a. 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 data flow information by an element of a lattice domain

- if $a < b$, then a is less precise than b
  - i.e., a is a conservative approximation to b
- top = most precise, best case info
- bottom = least precise, worst case info
- merge function = g.l.b. (meet) on lattice elements (the most precise element that’s a conservative approximation to both input elements)
- initial info for optimistic analysis (at least back edges): top

(Opposite up/down conventions used in PL semantics!)

Examples

Reaching definitions:

- an element:
- set of all elements:
- $\leq$:
- top:
- bottom:
- meet:

Reaching constants:

- an element:
- set of all elements:
- $\leq$:
- top:
- bottom:
- meet:

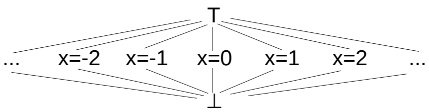
Some typical lattice domains

Powerset lattice: set of all subsets of a set *S*

- ordered by $\subseteq$ or $\supseteq$
- top & bottom = $\emptyset$ & *S*, or vice versa
- height = $|S|$ (infinite if *S* is infinite)
- "a collecting analysis"

A lifted set: a set of incomparable values, plus top & bottom

- e.g., reaching constants domain, for a particular variable:



- height = 3 (even though width is infinite!)

Two-point lattice: top and bottom

- computes a boolean property

Single-point lattice: just bottom

- trivial do-nothing analysis

Tuples of lattices

Often helpful to break down a complex lattice into a tuple of lattices, one per variable/stmt/... being analyzed

Formally: $D_T = <S_T, \leq_T> \ = \ (D = <S, \leq>)^N$

- $S_T = S_1 \times S_2 \times ... \times S_N$
  - element of tuple domain is a tuple of elements from each variable's domain
  - i^{th} component of tuple is info about i^{th} variable/stmt/...
- $<..., d_{1i}, ...> \ \leq_T \ <..., d_{2i}, ...> \ \equiv \ d_{1i} \leq d_{2i}, \forall i$
  - i.e. **pointwise** ordering
- meet: pointwise meet
- top: tuple of tops
- bottom: tuple of bottoms
- height(D_T) = *N* * height(D)

Powerset(*S*) lattice is isomorphic to a tuple of two-point lattices, one two-point lattice element per element of *S*

- i.e., a bit-vector!

Example: reaching constants

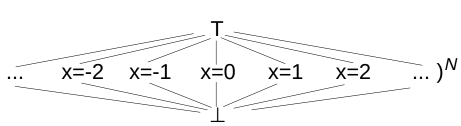
How to model reaching constants for all variables?

Informally:
each element is a set of the form $\{..., x \rightarrow k, ...\}$,
with at most one binding for *x*

One lattice model: a powerset of all $x \rightarrow k$ bindings

- S = pow($\{ x \rightarrow k \mid \forall x, \forall k \}$)
- $\leq \, = \, \subseteq$
- height?

Another lattice model:
N-tuple of 3-level constant prop. lattices,
for each of *N* variables

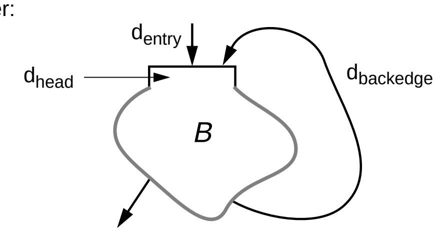
- 

- height?

Are they the same?
If not, which is better?

Analysis of loops in lattice model

Consider:



(Assume $B(d_{head})$ computes $d_{backedge}$)

Want solution to constraints:
$d_{head} = d_{entry} \cap d_{backedge}$
$d_{backedge} = B(d_{head})$

Let $F(d) = d_{entry} \cap B(d)$

Then want fixed-point of *F*:
$d_{head} = F(d_{head})$

Iterative analysis in lattice model

Iterative analysis computes fixed-point
by iterative approximation:

$F^0 = d_{entry} \cap T = d_{entry}$

$F^1 = d_{entry} \cap B(F^0) = F(F^0) = F(d_{entry})$

$F^2 = d_{entry} \cap B(F^1) = F(F^1) = F(F(F^0)) = F(F(d_{entry}))$

. . .

$F^k = d_{entry} \cap B(F^{k-1}) = F(F^{k-1}) = F(F(...(F(d_{entry}))...))$

until

$F^{k+1} = d_{entry} \cap B(F^k) = F(F^k) \quad = \quad F^k$

Is *k* finite?
If so, how big can it be?

Termination of iterative analysis

In general, *k* need not be finite

Sufficient conditions for finiteness:

- flow functions (e.g. *F*) are **monotonic**
- lattice is of finite height

A function *F* is monotonic iff:
$d_2 \le d_1 \Rightarrow F(d_2) \le F(d_1)$

- for application of DFA, this means that giving a flow function at least as conservative inputs ($d_2 \le d_1$) leads to at least as conservative outputs ($F(d_2) \le F(d_1)$)

For monotonic *F* over domain *D*, the maximum number of times that *F* can be applied to itself, starting w/ any element of D, w/o reaching fixed-point, is height(*D*)-1

- start at top of *D*
- for each application of F, either it's a fixed-point, or the result must go down at least one level in lattice
- eventually must hit a fixed-point (which will be the best fixed-point) or bottom (which is guaranteed to be a fixed-point), if *D* of finite height

Complexity of iterative analysis

How long does iterative analysis take?

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

What domain to use?

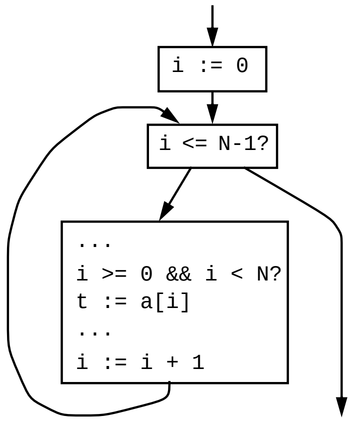
- what is its height?

What flow functions to use?

- are they monotonic?

Example

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



Widening operators

If domain is tall, then can introduce artificial generalizations (called **widenings**) when merging at loop heads

- ensure that only a finite number of widenings are possible
- not easy to design the “right” widening strategy

A generic worklist algorithm for lattice-theoretic DFA

Maintain a mapping from each program point to info at that point

- optimistically initialize all pp’s to T

Set initial pp’s (e.g. entry/exit point) to their correct values

Maintain a worklist of nodes whose flow functions need to be evaluated

- initialize with all nodes in graph
- include explicit meet & widening-meet nodes

While worklist nonempty do

Remove a node from worklist

Evaluate the node’s flow function,
given current info on predecessor/successor pp’s,
allowing it to change info on predecessor/successor pp’s

If any pp info changed, then put adjacent nodes on worklist (if not already there)

For faster analysis, want to follow topological order

- number nodes in topological order
- remove nodes from worklist in increasing topological order

Sharlit

A data flow analyzer generator [Tjiang & Hennessy 92]

- analogous to YACC

User writes basic primitives:

- control flow graph representation
  - nodes are instructions, not basic blocks
- domain (“flow value”) representation and key operations
  - `init`
  - `copy`
  - `is_equal`
  - `meet`
- flow functions for each kind of instruction
- action routines to optimize after analysis

Sharlit generates iterative dataflow analyzer from these pieces

+ easy to build, extend

– not highly efficient, so far...

Path compression

Can improve analysis efficiency by
summarizing effect of sequences of nodes

User can define path compression operations to collapse nodes together

- collapse linear sequence of nodes
  ⇒ summarizes effect of whole BB in a single node
  - presumes a fixed GEN/KILL bit-vector structure to be effective
- collapse trees ⇒ extended BB's
- collapse merges & loops as in interval analysis
  - use simplification to analyze reducible parts efficiently
  - use iteration to handle nonreducible parts

+ gets efficiency, preserves modularity & generality

– doesn't support data-dependent flow functions, cannot simulate optimizations during analysis

Performance results for code quality of generated optimizer, but not for compilation speed of optimizer

Vortex IDFA framework

Like Sharlit,
except a compiler library rather than a compiler-compiler

User defines a subclass of `AnalysisInfo` to represent elements of domain

- `copy`
- `merge` (lattice g.l.b. operator)
- `generalizing_merge` (g.l.b. with optional widening)
- `as_general_as` (lattice ≤ operator)

User invokes `traverse` to perform analysis:

```
cfg.traverse(direction, is_iterative?,
  initial_analysis_info,
  λ(rtl, info){ rtl.flow_fn(info) })
```

Flow function returns an `AnalysisResult`: one of

- keep instruction and continue analysis w/ updated info(s)
- delete instruction/constant-fold branch
- replace instruction with instruction or subgraph

`ComposedAnalysis` supports running multiple analyses interleaved at each instruction

Features of Vortex IDFA

Big idea: separate analyses and transformations, make framework compose them appropriately

- don't have to simulate the effect of transformations during analysis
- can run analyses in parallel if each provides opportunities for the other
  - sometimes can achieve strictly better results this way than if run separately in a loop
- more general transformations supported (e.g. inlining) than Sharlit

Exploit inheritance & closures

Analysis speed is not stressed

- no path compression
- no "compilation" of analysis with framework

[Vortex's interprocedural analysis support discussed later]