Lattice-Theoretic Data Flow Analysis

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: 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

- convenient to count edges
- infinitely large lattice can still have finite height

Top (T) = unique greatest element of S, if it exists
Bottom ($\perp$) = unique least element of S, if it exists

Lattice models in data flow analysis

Data flow info at a prog. pt. modeled by an *element* of a lattice

- our convention: 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

(Reverse less precise/more precise conventions used in PL semantics & abstract interpretation!)

Examples

Reaching definitions:

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

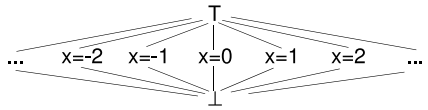
Reaching constants:

- 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 = 2 (even though width may be infinite!)

Two-point lattice: top and bottom
- represents a boolean property

Single-point lattice: top = bottom
- trivial do-nothing analysis

Tuples of lattices

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

Formally: D = <S, ≤> = $(D_i = \langle S_i, \leq_i \rangle)^N$
- $S = S_1 \times S_2 \times ... \times S_N$
  - element of tuple domain is a tuple of elements drawn from each component domain
  - e.g., i^{th} component of tuple is info about i^{th} variable/stmt/...
- $\langle ..., d_{1i}, ... \rangle \leq \langle ..., d_{2i}, ... \rangle \equiv d_{1i} \leq_i d_{2i}, \forall i$
  - i.e. **pointwise** ordering
- meet: pointwise meet
- top: tuple of tops
- bottom: tuple of bottoms
- height(D) = height(D_1) × ... × height(D_N)

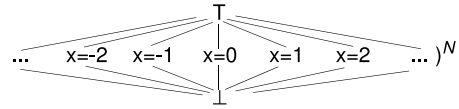
Powerset(*S*) lattice is isomorphic to a tuple of two-point lattices, one two-point lattice per element of *S*
- i.e., a bit-vector!

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

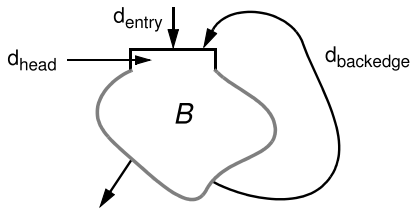
One lattice model: a powerset of all $x \rightarrow k$ bindings
- S = pow({ $x \rightarrow k$ | $\forall x, \forall k$ })
- ≤ = $\subseteq$
- height?

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

- height?

Analysis of loops in lattice model

Consider:



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

Want solution to constraints:

$d_{head} = d_{entry} \cap d_{backedge}$ [∩ means meet]

$d_{backedge} = B(d_{head})$

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, beginning with $d_{backedge}=T$:

$f_0 = d_{entry} \cap T = d_{entry}$

$f_1 = d_{entry} \cap B(f_0) = F(f_0) = F(d_{entry})$

$f_2 = d_{entry} \cap B(f_1) = F(f_1) = F(F(f_0)) = F(F(d_{entry})) = F^2(d_{entry})$

. . .

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

until

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

Does a finite k exist?
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 DFA, 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)

- 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
- if D of finite height, eventually must hit a fixed-point
  - bottom is always a fixed-point, by monotonicity

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

Precision of iterative analysis

Iterative analysis finds a fixed-point f of F, $f = F(f)$

Is it the best fixed-point?
I.e., is it the case that, $\forall f_i$ s.t. $f_i = F(f_i)$, $f_i \le f$?

Answer: yes!

Proof:

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?

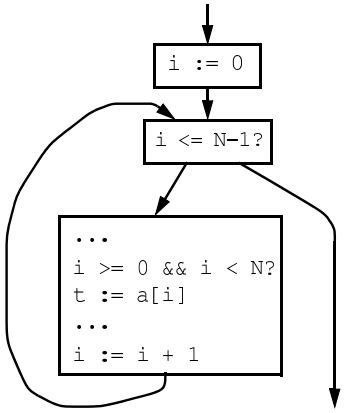
- what is its height?

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 (merge) &
  widening-meet (loop-head-merge) 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 successor(predecessor) pp’s
If any pp info changed, put successor(predecessor) nodes on worklist (if not already there)

For faster analysis, want to follow topological order

- number nodes in forward(backward) topological order
- remove nodes from worklist in increasing topological order

Whirlwind dataflow analysis engine

Client defines subclass(es) of `LatticeElmt` (a subclass of `AnalysisInfo`) to represent elements of domain

- <= (lattice ≤ operator)
- `meet` (lattice g.l.b. operator)

Client picks a subclass of `AnalysisGraph` to specify the graph over which to analyze

- {`Forward`,`Backward`}{`CFG`,`DFG`}`AnalysisGraph`

Client defines a subclass of `Analysis` that describes the analysis

- `top_analysis_info` (the top `LatticeElmt` instance)
- `analyze(Analysis, AnalysisGraph, TargetIRNode, indexed[LatticeElmt] ):AnalysisAction` (the flow function)
  - typically many `analyze` multimethods dispatching on different `TargetIRNode` subclasses

Client invokes `analyze_and_transform( Analysis,AnalysisGraph,indexd[AnalysisInfo])` to run the analysis and do all the transformations

- wrapper functions used in practice, e.g. `do_optimization`

Analysis actions

The result of the `analyze` flow function on an `IRNode` is either

- `ContinueAnalysisAction`: propagate a resulting `AnalysisInfo` along successor edge(s)
- `ReplaceAnalysisAction`: replace the `IRNode` with some other sub-`AnalysisGraph`, and restart analysis

`ReplaceAnalysisAction` specifies the transformation to perform as a result of analysis

Also implicitly specifies how to *simulate* the transformation during iterative analysis

- the engine transparently analyzes the replacement graph in lieu of the replaced `IRNode`, to simulate what would happen if the transformation were done

Composed analyses

Whirlwind allows several dataflow analyses to be performed "in parallel"

- interleaved at each `IRNode` operation

If one analysis chooses a transformation, others are reevaluated on the replacement subgraph

- allows improvements of one analysis to improve quality of other analyses, without any explicit accounting in them

Client defines each component analysis as subclass of `ComposableAnalysis`

Client defines a composition of analyses as subclass of {`Forward`,`Backward`}`ComposedAnalysis`

`ComposedAnalysis` is just a regular analysis whose `analyze` flow function invokes each of the component analyses' flow functions in turn

[Lerner, Grove, Chambers, POPL '02]

Features of Whirlwind's dataflow analysis engine

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
- quite drastic transformations supported (e.g. inlining, branch folding) during analysis
- no non-local transformations (e.g. code motion) supported

Makes no sacrifices of precision for speed

- has few speed-related optimizations

Soundness of Data Flow Analysis

We'd like to convince ourselves, even prove formally, that our dataflow analysis is correct, i.e., sound, with respect to some intended uses

We need two things:

- a reference *concrete* semantics that defines the "truth," against which we compare our *abstract* semantics
  - including a *concrete domain* of information at program points against which we compare our *abstract domain* of analysis results at program points
- an *abstraction relation* that specifies when an abstract domain element conservatively approximates a concrete domain element, for our intended uses

(Developed in the framework of **abstract interpretation** by Cousot & Cousot [POPL '77, '79])

Concrete semantics

Many ways to define the semantics of a programming language

A good way for our purposes is *small-step operational semantics*, i.e., a set of transition rules

An example transition rule:

$\langle pp_{in}, mem_{in} \rangle \rightarrow_{S:\ X := Y+Z} \langle pp_{out}, mem_{out} \rangle$

where $\{pp_{in}\}$ = pred-pts(S)

$\{pp_{out}\}$ = succ-pts(S)

$mem_{out} = mem_{in}[X \rightarrow (mem_{in}(Y) + mem_{in}(Z))]$

"if execution reaches program point pp_{in} with memory state mem_{in}, and the instruction S after that program point is of the form $X := Y+Z$, then program execution may 'step' to program point pp_{out} with memory state mem_{out}."

These small-step rules are just (concrete) flow functions!

- but the info being "propagated" is the whole state of the computation (and the outside world, perhaps)
- but control flow is more explicit, to account for which way execution proceeds after branches

Traces

Concrete execution of a whole program is a *trace*

- sequence of <*pp*,*mem*> pairs, starting from the initial program entry point and memory state, following the concrete flow functions, until reaching final <*pp*,*mem*> for which no transition is defined
- could be infinitely long

[Aside:

If convenient, we can collapse traces onto the control flow graph, storing not a sequence of pairs but rather a map from each program point to the set of all memories that occur in the trace at that program point, called the *collecting (concrete) semantics*

{ (pp→mems) |
mems = { mem' | <pp,mem'> ∈ Trace } }

]

Abstraction relation

Now we have concrete information (memories) and abstract information (domain elements computed by our analysis). When does the abstract information safely, possibly conservatively, characterize the concrete information?

Depends on the use/intention of the abstract info

Define this using an *abstraction relation* α:

For concrete info c and abstract info a, $(c,a) \in \alpha$ iff a is a safe approximation of c

E.g., for constant propagation [where $d_{CP} \subseteq$ Var×Const]:

$(mem, d_{CP}) \in \alpha_{CP} \Leftrightarrow$

$\forall (var \rightarrow const) \in d_{CP}.\ mem(var) = const$

(Could define α as a relation between whole traces and abstract info, to allow the abstract info to approximate history- or future-sensitive info, e.g. for reaching defs or live variables)

Local and global soundess

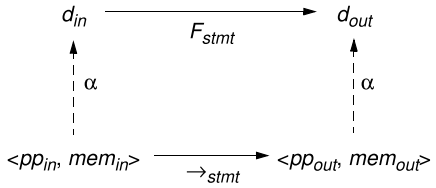
Lemma (Local soundness of analysis A = <α, F>).

if $\langle pp_{in}, mem_{in}\rangle \rightarrow_{stmt} \langle pp_{out}, mem_{out}\rangle$

and $(mem_{in}, d_{in}) \in \alpha$

and $F_{stmt}(d_{in}) = d_{out}$

then $(mem_{out}, d_{out}) \in \alpha$



- prove this by examining each F flow function case

Theorem (Global soundess of analysis A = <α, F>).
If we start the abstract analysis with safe abstract info at the first program point, the abstract analysis will compute safe abstract info at each program point in the trace.

- by induction over the trace, using local soundness lemma
- proof is independent of the actual analysis!

Rhodium

Specify dataflow analyses in a specialized declarative language

+ easier to specify optimization than raw Diesel/C++/... code
+ allow mechanical proof of correctness of optimizations!
+ allow mechanical composition & compilation down to efficient code

A prototype implemented in Whirlwind

- analyses & transforms handle a C-like subset of full WIL
- correctness checked automatically!
- working on *inferring* flow functions automatically from domain definitions!
- execution engine *very* slow, currently...

[Lerner, Millstein, Chambers, PLDI '03;
Lerner, Millstein, Rice, Chambers, POPL '05;
Rice, Lerner, Chambers, COCV '05]