Static Program Analysis
Part 3 – lattices and fixpoints

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University
Flow-sensitivity

• Type checking is (usually) flow-insensitive:
  – statements may be permuted without affecting typability
  – constraints are naturally generated from AST nodes

• Other analyses must be flow-sensitive:
  – the order of statements affects the results
  – constraints are naturally generated from \textit{control flow graph nodes}
Sign analysis

• Determine the sign (+, −, 0) of all expressions
• The Sign lattice:

  \[ + \quad - \quad 0 \]

  \[ \top \quad \bot \]

  “any number”

  “not of type number” (or, “unreachable code”)

• States are modeled by the map lattice \( Vars \to \text{Sign} \)
  where \( Vars \) is the set of variables in the program

Implementation: TIP/src/tip/analysis/SignAnalysis.scala
Generating constraints

```plaintext
1 var a,b;
2 a = 42;
3 b = a + input;
4 a = a - b;
```

$x_1 = [a \mapsto T, b \mapsto T]$

$x_2 = x_1[a \mapsto +]$  

$x_3 = x_2[b \mapsto x_2(a) + T]$  

$x_4 = x_3[a \mapsto x_3(a) - x_3(b)]$
Sign analysis constraints

• The variable $[v]$ denotes a map that gives the sign value for all variables at the program point after node $v$

• For variable declarations:
  
  \[
  [\text{var } x_1, \ldots, x_n] = JOIN(v)[x_1 \mapsto T, \ldots, x_n \mapsto T]
  \]

• For assignments:

  \[
  [x = E] = JOIN(v)[x \mapsto eval(JOIN(v),E)]
  \]

• For all other nodes:

  \[
  [v] = JOIN(v)
  \]

where $JOIN(v) = \bigsqcup [w]_{w \in pred(v)}$ combines information from predecessors (explained later…)
Evaluating signs

- The `eval` function is an *abstract evaluation*:
  - `eval(\sigma, x) = \sigma(x)`
  - `eval(\sigma, \text{intconst}) = \text{sign}(\text{intconst})`
  - `eval(\sigma, E_1 \text{ op } E_2) = \text{op}(eval(\sigma, E_1), eval(\sigma, E_2))`

- \(\sigma\): \(\text{Vars} \rightarrow \text{Sign}\) is an abstract state

- The `sign` function gives the sign of an integer

- The `\text{op}` function is an abstract evaluation of the given operator
Abstract operators

(assuming the subset of TIP with only integer values)
Increasing precision

- Some loss of information:
  - \((2>0)==1\) is analyzed as \(\top\)
  - \(+/-+\) is analyzed as \(\top\), since e.g. \(\frac{1}{2}\) is rounded down

- Use a richer lattice for better precision:

- Abstract operators are now \(8\times8\) tables
Partial orders

• Given a set $S$, a partial order $\sqsubseteq$ is a binary relation on $S$ that satisfies:
  
  – reflexivity: $\forall x \in S: x \sqsubseteq x$
  
  – transitivity: $\forall x, y, z \in S: x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  
  – anti-symmetry: $\forall x, y \in S: x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$

• Can be illustrated by a Hasse diagram (if finite)
Upper and lower bounds

• Let $X \subseteq S$ be a subset

• We say that $y \in S$ is an upper bound ($X \subseteq y$) when
  \[ \forall x \in X: x \subseteq y \]

• We say that $y \in S$ is a lower bound ($y \subseteq X$) when
  \[ \forall x \in X: y \subseteq x \]

• A least upper bound $\bigcup X$ is defined by
  \[ X \subseteq \bigcup X \land \forall y \in S: X \subseteq y \Rightarrow \bigcup X \subseteq y \]

• A greatest lower bound $\bigwedge X$ is defined by
  \[ \bigwedge X \subseteq X \land \forall y \in S: y \subseteq X \Rightarrow y \subseteq \bigwedge X \]
Lattices

• A *lattice* is a partial order where $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in S$ \hspace{1cm} ($x \sqcup y$ is notation for $\sqcup \{x, y\}$)

• A *complete lattice* is a partial order where $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$

• A complete lattice must have
  – a unique largest element, $\top = \sqcup S$ \hspace{1cm} *(exercise)*
  – a unique smallest element, $\bot = \sqcap S$

• A finite lattice is complete if $\top$ and $\bot$ exist

Implementation: TIP/src/tip/lattices/
These partial orders are lattices
These partial orders are *not* lattices
The powerset lattice

• Every finite set $A$ defines a complete lattice $(\mathcal{P}(A), \subseteq)$ where
  
  - $\perp = \emptyset$
  
  - $\top = A$
  
  - $x \sqcup y = x \cup y$
  
  - $x \sqcap y = x \cap y$

\[
\begin{array}{c}
\{0,1,2,3\} \\
\{0,1,2\} \quad \{0,1,3\} \quad \{0,2,3\} \quad \{1,2,3\} \\
\{0,1\} \quad \{0,2\} \quad \{0,3\} \quad \{1,2\} \quad \{1,3\} \quad \{2,3\} \\
\{0\} \quad \{1\} \quad \{2\} \quad \{3\} \\
\{\} \quad \text{for } A = \{0,1,2,3\}
\end{array}
\]
Lattice height

- The *height* of a lattice is the length of the longest path from $\bot$ to $\top$
- The lattice $(\mathcal{P}(A), \subseteq)$ has height $|A|$ for $A = \{0, 1, 2, 3\}$
Map lattice

- If A is a set and L is a complete lattice, then we obtain a complete lattice called a map lattice:

\[ A \rightarrow L = \{ [a_1 \mapsto x_1, a_2 \mapsto x_2, ...] \mid A = \{a_1, a_2, ...\} \land x_1, x_2, ... \in L \} \]

ordered pointwise

- \( \sqcup \) and \( \sqcap \) can be computed pointwise

- \( \text{height}(A \rightarrow L) = |A| \cdot \text{height}(L) \)

Example: \( A \rightarrow L \) where
- A is the set of program variables
- L is the \textit{Sign} lattice
Product lattice

• If $L_1$, $L_2$, ..., $L_n$ are complete lattices, then so is the *product*:

$$L_1 \times L_2 \times ... \times L_n = \{ (x_1, x_2, ..., x_n) \mid x_i \in L_i \}$$

where $\sqsubseteq$ is defined pointwise

• Note that $\sqcup$ and $\sqcap$ can be computed pointwise

• $height(L_1 \times L_2 \times ... \times L_n) = height(L_1) + ... + height(L_n)$

Example:
each $L_i$ is the map lattice $A \to L$ from the previous slide, and $n$ is the number of CFG nodes
Flat lattice

• If A is a set, then $\text{flat}(A)$ is a complete lattice:

\[
\begin{array}{c}
\top \\
\downarrow \\
a_1 \quad a_2 \quad ... \quad a_n \\
\downarrow \\
\bot
\end{array}
\]

• $\text{height}(\text{flat}(A)) = 2$
If $L$ is a complete lattice, then so is $lift(L)$, which is:

- $height(lift(L)) = height(L) + 1$
The variable $⟦v⟧$ denotes a map that gives the sign value for all variables at the program point after node $v$

$⟦v⟧ ∈ States$ where $States = Vars → Sign$

For variable declarations:

$⟦\text{var } x_1, \ldots, x_n⟧ = \text{JOIN}(v)[x_1 \mapsto T, \ldots, x_n \mapsto T]$

For assignments:

$⟦x = E⟧ = \text{JOIN}(v)[x \mapsto \text{eval}(\text{JOIN}(v),E)]$

For all other nodes:

$⟦v⟧ = \text{JOIN}(v)$

where $\text{JOIN}(v) = \bigcup_{w ∈ \text{pred}(v)} ⟦w⟧$ combines information from predecessors
Generating constraints

```
var a, b, c;
a = 42;
b = 87;
if (input) {
c    c = a + b;
} else {
c    c = a - b;
}
```

\[
\begin{align*}
\llbracket entry \rrbracket &= \bot \\
\llbracket \text{var } a, b, c \rrbracket &= \llbracket entry \rrbracket[a \mapsto T, b \mapsto T, c \mapsto T] \\
\llbracket a = 42 \rrbracket &= \llbracket \text{var } a, b, c \rrbracket[a \mapsto +] \\
\llbracket b = 87 \rrbracket &= \llbracket a = 42 \rrbracket[b \mapsto +] \\
\llbracket \text{input} \rrbracket &= \llbracket b = 87 \rrbracket \\
\llbracket c = a + b \rrbracket &= \llbracket \text{input} \rrbracket[c \mapsto \llbracket \text{input} \rrbracket(a) + \llbracket \text{input} \rrbracket(b)] \\
\llbracket c = a - b \rrbracket &= \llbracket \text{input} \rrbracket[c \mapsto \llbracket \text{input} \rrbracket(a) - \llbracket \text{input} \rrbracket(b)] \\
\llbracket \text{exit} \rrbracket &= \llbracket c = a + b \rrbracket \cup \llbracket c = a - b \rrbracket
\end{align*}
\]
Constraints

• From the program being analyzed, we have constraint variables $x_1, ..., x_n \in L$ and a collection of constraints:

\[
\begin{align*}
  x_1 &= f_1(x_1, ..., x_n) \\
  x_2 &= f_2(x_1, ..., x_n) \\
  &\quad \vdots \\
  x_n &= f_n(x_1, ..., x_n)
\end{align*}
\]

• These can be collected into a single function $f: L^n \rightarrow L^n$:

$$f(x_1, ..., x_n) = (f_1(x_1, ..., x_n), ..., f_n(x_1, ..., x_n))$$

• How do we find the least (i.e. most precise) value of $x_1, ..., x_n$ such that $(x_1, ..., x_n) = f(x_1, ..., x_n)$ (if that exists)?

Note that $L^n$ is a product lattice.
Monotone functions

- A function $f: L \to L$ is monotone when
  \[ \forall x, y \in L: x \sqsubseteq y \implies f(x) \sqsubseteq f(y) \]
- A function with several arguments is monotone if it is monotone in each argument
- Monotone functions are closed under composition
- As functions, $\sqcup$ and $\sqcap$ are both monotone
- $x \sqsubseteq y$ can be interpreted as “$x$ is at least as precise as $y$”
- When $f$ is monotone: “more precise input cannot lead to less precise output”

(exercises)
Monotonicity for the sign analysis

- The $\sqcup$ operator and map updates are monotone
- Compositions preserve monotonicity
- Are the abstract operators monotone?
- Can be verified by a tedious inspection:
  - $\forall x, y, x' \in L: x \sqsubseteq x' \Rightarrow x \overline{\circ p} y \sqsubseteq x' \overline{\circ p} y$
  - $\forall x, y, y' \in L: y \sqsubseteq y' \Rightarrow x \overline{\circ p} y \sqsubseteq x \overline{\circ p} y'$

Example, constraints for assignments:
$\llbracket x = E \rrbracket = \text{JOIN}(v)[x \mapsto \text{eval}(\text{JOIN}(v),E)]$
Kleene’s fixed-point theorem

\( x \in L \) is a \textit{fixed point} of \( f : L \to L \) iff \( f(x) = x \)

In a complete lattice with finite height, every monotone function \( f \) has a \textit{unique least fixed-point}:

\[
\text{fix}(f) = \bigsqcup_{i \geq 0} f^i(\perp)
\]
Proof of existence

- Clearly, \( \bot \sqsubseteq f(\bot) \)
- Since \( f \) is monotone, we also have \( f(\bot) \sqsubseteq f^2(\bot) \)
- By induction, \( f^i(\bot) \sqsubseteq f^{i+1}(\bot) \)
- This means that
  \[
  \bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \ f^i(\bot) \ldots
  \]
is an increasing chain
- \( L \) has finite height, so for some \( k \): \( f^k(\bot) = f^{k+1}(\bot) \)
- If \( x \sqsubseteq y \) then \( x \sqcup y = y \) \hspace{1cm} (exercise)
- So \( \text{fix}(f) = f^k(\bot) \)
Proof of unique least

• Assume that \( x \) is another fixed-point: \( x = f(x) \)
• Clearly, \( \bot \sqsubseteq x \)
• By induction and monotonicity, \( f^i(\bot) \sqsubseteq f^i(x) = x \)
• In particular, \( \text{fix}(f) = f^k(\bot) \sqsubseteq x \), i.e. \( \text{fix}(f) \) is least

• Uniqueness then follows from anti-symmetry
Computing fixed-points

The time complexity of $\text{fix}(f)$ depends on:

- the height of the lattice
- the cost of computing $f$
- the cost of testing equality

$x = \bot$
do {
    $t = x$
    $x = f(x)$
} while ($x \neq t$);

Implementation: TIP/src/tip/solvers/FixpointSolvers.scala
Summary: lattice equations

- Let $L$ be a complete lattice with finite height

- A equation system is of the form:

  $$x_1 = f_1(x_1, \ldots, x_n)$$
  $$x_2 = f_2(x_1, \ldots, x_n)$$
  $$\ldots$$
  $$x_n = f_n(x_1, \ldots, x_n)$$

  where $x_i$ are variables and each $f_i: L^n \rightarrow L$ is monotone

- Note that $L^n$ is a product lattice
Solving equations

• Every equation system has a *unique least solution*, which is the least fixed-point of the function $f: \mathbb{L}^n \rightarrow \mathbb{L}^n$ defined by

$$f(x_1,\ldots,x_n) = (f_1(x_1,\ldots,x_n), \ldots, f_n(x_1,\ldots,x_n))$$

• A solution is always a fixed-point (for any kind of equation)

• The least one is the most precise
Solving inequations

• A *inequation system* is of the form

\[
\begin{align*}
x_1 & \sqsubseteq f_1(x_1, \ldots, x_n) & x_1 & \sqsupseteq f_1(x_1, \ldots, x_n) \\
x_2 & \sqsubseteq f_2(x_1, \ldots, x_n) & x_2 & \sqsupseteq f_2(x_1, \ldots, x_n) \\
\vdots & & \vdots \\
x_n & \sqsubseteq f_n(x_1, \ldots, x_n) & x_n & \sqsupseteq f_n(x_1, \ldots, x_n)
\end{align*}
\]

or

• Can be solved by exploiting the facts that

\[
\begin{align*}
x \sqsubseteq y & \iff x = x \sqcap y \\
\text{and} \\
x \sqsupseteq y & \iff x = x \sqcup y
\end{align*}
\]
Monotone frameworks


- A CFG to be analyzed, nodes Nodes = \{v_1, v_2, ..., v_n\}
- A finite-height complete lattice L of possible answers
  - fixed or parametrized by the given program
- A constraint variable \([v] \in L\) for every CFG node v
- A dataflow constraint for each syntactic construct
  - relates the value of \([v]\) to the variables for other nodes
  - typically a node is related to its neighbors
  - the constraints must be monotone functions:
    \([v_i] = f_i([v_1], [v_2], ..., [v_n])\)
Monotone frameworks

• Extract all constraints for the CFG

• Solve constraints using the fixed-point algorithm:
  – we work in the lattice $L^n$ where $L$ is a lattice describing abstract states
  – computing the least fixed-point of the combined function:
    \[ f(x_1,\ldots,x_n) = (f_1(x_1,\ldots,x_n), \ldots, f_n(x_1,\ldots,x_n)) \]

• This solution gives an answer from $L$ for each CFG node
Generating and solving constraints

Conceptually, we separate constraint generation from constraint solving, but in implementations, the two stages are typically interleaved.
Lattice points as answers

the trivial, useless answer

our answer (the least fixed-point)

safe answers

unsafe answers

the true answer

Conservative approximation...
The naive algorithm

\[ x = (\bot, \bot, \ldots, \bot); \]
\[ \text{do } \{ \]
\[ \quad t = x; \]
\[ \quad x = f(x); \]
\[ \} \text{ while } (x \neq t); \]

• Correctness ensured by the fixed point theorem
• Does not exploit any special structure of \( L^n \) or \( f \)
  (i.e. \( x \in L^n \) and \( f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)) \))

Implementation: SimpleFixpointSolver
Example: sign analysis

```plaintext
ite(n) {
    var f;
    f = 1;
    while (n>0) {
        f = f*n;
        n = n-1;
    }
    return f;
}
```

(We shall later see how to improve precision for the loop condition)
The naive algorithm

Computing each new entry is done using the previous column

- Without using the entries in the current column that have already been computed!
- And many entries are likely unchanged from one column to the next!
Chaotic iteration

Recall that $f(x_1,\ldots,x_n) = (f_1(x_1,\ldots,x_n), \ldots, f_n(x_1,\ldots,x_n))$

$x_1 = \perp; \ldots x_n = \perp$

while $((x_1,\ldots,x_n) \neq f(x_1,\ldots,x_n))$ {
    pick $i$ nondeterministically such that $x_i \neq f_i(x_1, \ldots, x_n)$
    $x_i = f_i(x_1, \ldots, x_n)$
}

We now exploit the special structure of $L^n$
– may require a higher number of iterations,
but less work in each iteration
Correctness of chaotic iteration

• Let $x^j$ be the value of $x=(x_1, \ldots, x_n)$ in the $j$’th iteration of the naive algorithm
• Let $\underline{x}^j$ be the value of $x=(x_1, \ldots, x_n)$ in the $j$’th iteration of the chaotic iteration algorithm
• By induction in $j$, show $\forall j: \underline{x}^j \subseteq x^j$
• Chaotic iteration eventually terminates at a fixed point
• It must be identical to the result of the naive algorithm since that is the least fixed point
Towards a practical algorithm

• Computing $\exists i : \ldots$ in chaotic iteration is not practical

• Idea: predict $i$ from the analysis and the structure of the program!

• Example:
  In sign analysis, when we have processed a CFG node $v$, process $\text{succ}(v)$ next
The worklist algorithm (1/2)

• Essentially a specialization of chaotic iteration that exploits the special structure of f

• Most right-hand sides of \( f_i \) are quite sparse:
  – constraints on CFG nodes do not involve all others

• Use a map:

\[
dep: \text{Nodes} \rightarrow 2^{\text{Nodes}}
\]

that for \( v \in \text{Nodes} \) gives the set of nodes (i.e. constraint variables) \( w \) where \( v \) occurs on the right-hand side of the constraint for \( w \)
The worklist algorithm (2/2)

\[ x_1 = \perp; \ldots \ x_n = \perp; \]
\[ W = \{v_1, \ldots, v_n\}; \]
\[ \textbf{while} \ (W \neq \emptyset) \ {\} \]
\[ \quad v_i = W.\text{removeNext}(); \]
\[ \quad y = f_i(x_1, \ldots, x_n); \]
\[ \quad \textbf{if} \ (y \neq x_i) \ {\} \]
\[ \quad \quad \textbf{for} \ (v_j \in \text{dep}(v_i)) \ W.\text{add}(v_j); \]
\[ \quad x_i = y; \]
\[ \} \]

Implementation: SimpleWorklistFixpointSolver
Further improvements

• Represent the worklist as a priority queue
  – find clever heuristics for priorities

• Look at the graph of dependency edges:
  – build strongly-connected components
  – solve constraints bottom-up in the resulting DAG
Transfer functions

• The constraint functions in dataflow analysis usually have this structure:
\[
\llbracket v \rrbracket = t_v(JOIN(v))
\]
where \( t_v : \text{States} \rightarrow \text{States} \) is called the \textit{transfer function} for \( v \)

• Example:
\[
\llbracket x = E \rrbracket = JOIN(v)[x \mapsto eval(JOIN(v),E)]
= t_v(JOIN(v))
\]
where
\[
t_v(s) = s[x \mapsto eval(s,E)]
\]
Sign Analysis, continued...

- Another improvement of the worklist algorithm:
  - only add the entry node to the worklist initially
  - then let dataflow propagate through the program according to the constraints...

- Now, what if the constraint rule for variable declarations was:
  
  \[
  \llbracket \text{var } x_1, \ldots, x_n \rrbracket = \text{JOIN}(v)[x_1 \mapsto \perp, \ldots, x_n \mapsto \perp]
  \]

  (would make sense if we treat “uninitialized” as “no value” instead of “any value”)

- Problem: iteration would stop before the fixpoint!
- Solution: replace \( \text{Vars} \rightarrow \text{Sign} \) by \( \text{lift}(\text{Vars} \rightarrow \text{Sign}) \)
  
  (allows us to distinguish between “unreachable” and “all variables are non-integers”)
- This trick is also useful for context-sensitive analysis! (later...)

Implementation: WorklistFixpointSolverWithReachability, MapLiftLatticeSolver