Applied Static Analysis

Data Flow Analysis

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

Lattice Theory

Many static analyses are based on the mathematical theory of lattices.

The lattice put the facts (often, but not always, sets) computed by an analysis in a well-defined partial order.

Analysis are often **well-defined** functions over lattices and can then be combined and reasoned about.

Example: Sign Analysis

- Let's assume that we want to compute the sign of an integer value. The analysis should only return the information is definite. I.e.,

- Instead of computing with concrete values, our analysis performs it computations using abstract values:
  - positive (+)
  - negative (-)
  - zero

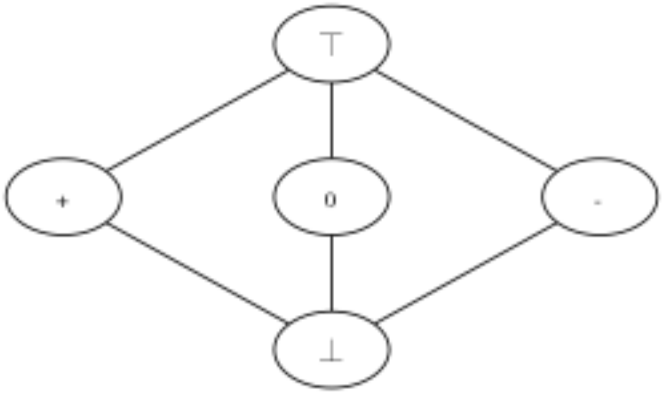
- Additionally, we have to add an abstract value $\top$ that represents the fact that we don't know the sign of the value.

- Values that are not initialized are represented using $\bot$.

$\top$ is called *top*. (The least precise information. *The sound over-approximation*.)
$\bot$ is called *bottom*. (The most precise information.)

Example: Sign Analysis - the lattice

The lattice for the previous domain is:



The ordering reflects that ⊤ reflects all types of integer values.

Example: Sign Analysis - example program

```
def select(c : Boolean): Int = {
    val a = 42
    val b = 333
    var x = 0;
    if (c)
        x = a + b;
    else
        x = a - b;
    x
}
```

A possible result of the analysis could be that `a` and `b` are always positive; x is either positive or negative ($\perp$).

Partial Orderings

- a partial ordering is a relation $\sqsubseteq: L \times L \to \{true, false\}$, which
  - is reflexiv: $\forall l : l \sqsubseteq l$
  - is transitive: $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
  - is anti-symmetric: $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$
- a partially ordered set $(L, \sqsubseteq)$ is a set L equipped with a partial ordering $\sqsubseteq$

When $x \sqsubseteq y$ we say x is at least as precise as y or y over-approximates x/y is an over-approximation of y.

Upper Bounds

- for $Y \subseteq L$ and $l \in L$
  - l is an upper bound of Y, if $\forall l' \in Y : l' \sqsubseteq l$
  - l is a **least upper bound** of Y, if $l \sqsubseteq l_0$ whenever l_0 is also an upper bound of Y
- if a least upper bound exists, it is unique ($\sqsubseteq$ is anti-symmetric)
- the least upper bound of Y is denoted $\bigsqcup Y$
  we write: $l1 \sqcup l2$ for $\bigsqcup \{l1, l2\}$

$\sqcup$ is also called the join operator.

Lower Bounds

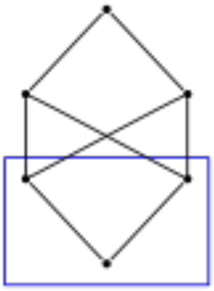
- for $Y \subseteq L$ and $l \in L$
  - l is a lower bound of Y, if $\forall l' \in Y : l \sqsubseteq l'$
  - l is a **greatest lower bound** of Y, if $l_0 \sqsubseteq l$ whenerver l_0 is also a lower bound of Y

- if a greatest lower bound exists, it is unique (⊑ is anti-symmetric)
- the greatest lower bound of Y is denoted $\bigsqcap Y$
  we write: $l1 \sqcap l2$ for $\bigsqcap\{l1, l2\}$

⊓ is also called the meet operator.

Upper/Lower Bounds

A subset Y of a partially ordered set L need not have least upper or greatest lower bounds.



Here, the subset is depicted in blue.

(complete) Lattice

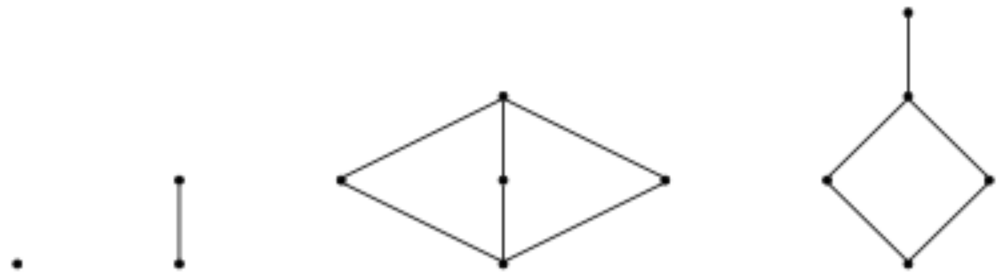
- complete Lattice $L = (L, \sqsubseteq, \sqcap, \bigsqcup, \top, \bot)$
- is a partially ordered set $(L, \sqsubseteq)$ such that each subset Y has a greatest lower bound and a least upper bound.
  - $\bot = \bigsqcup \emptyset = \sqcap L$
  - $\top = \sqcap \emptyset = \bigsqcup L$

A lattice must have a unique largest element $\top$ and a unique smallest element $\bot$.

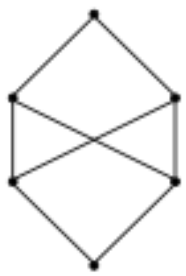
The least upper bound (the greatest lower bound) of a set $Y \subseteq L$ can contain the least (the greatest) element $l \in L$

[.hide-footer]

Valid lattices:



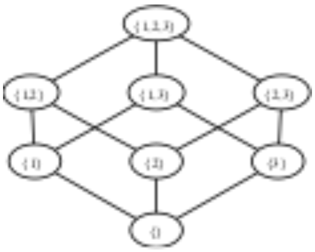
No lattice:



Do ask yourself why the lower diagram is not a lattice?

(complete) Lattice - example

Example $(\mathcal{P}(S), \subseteq)$, $S = \{1, 2, 3\}$



The above diagram is called a *Hasse* diagram.

Every finite set S defines a lattice $(\mathcal{P}S, \subseteq)$ where $\perp = \emptyset$ and $\top = S$, $x \bigsqcup y = x \cup y$, and $x \sqcap y = x \cap y$

A Hasse diagram is a graphical representation of the relation of elements of a partially ordered set (poset).

Height of a lattice

The length of the longest path from $\perp$ to $\top$.

In general, the powerset lattice has height $|S|$.

The height of the previous lattice is 3.

Closure Properties

Construction of complete lattices:

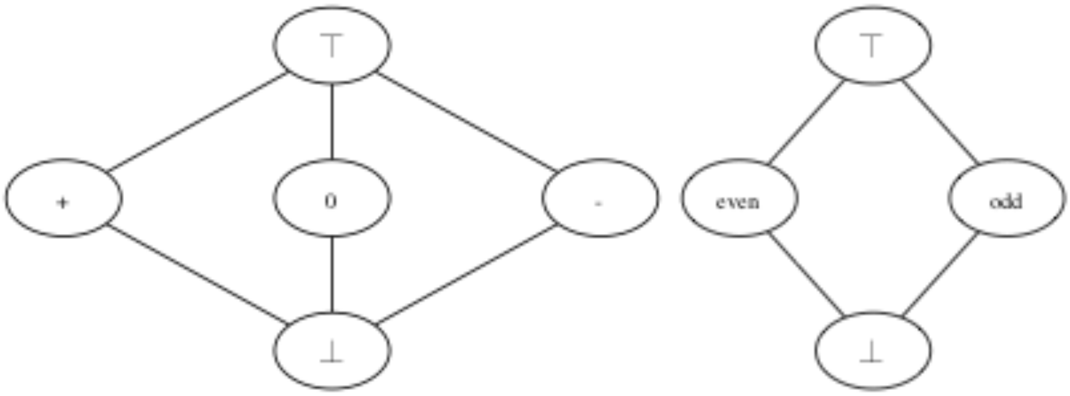
If $L_1, L_2, \ldots, L_n$ are lattices with finite height, then so is the (cartesian) product:

$$L_1 \times L_2 \times \cdots \times L_n = \\ (x_1, x_2, \ldots, x_n) | X_i \in L_i$$

$$height(L_1 \times \cdots \times L_n) = height(L_1) + \cdots + height(L_n)$$

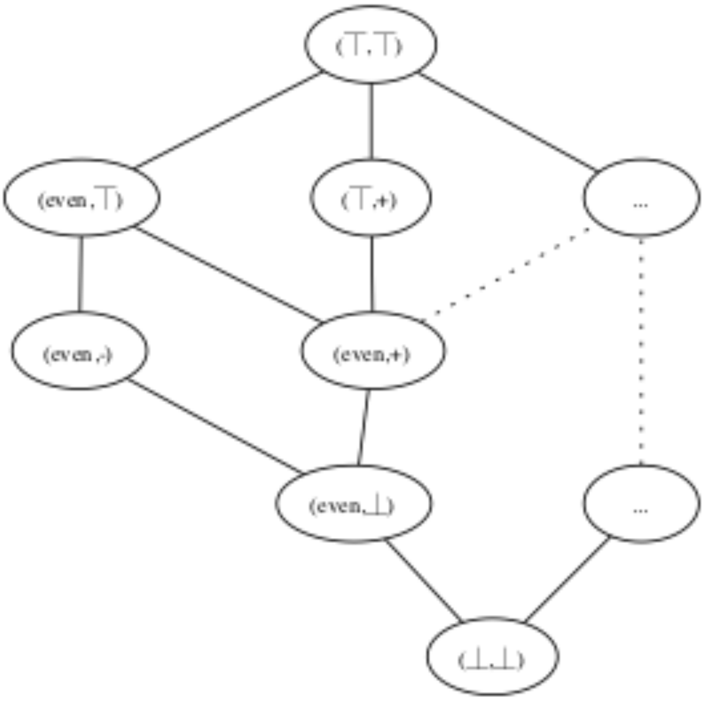
$\sqsubseteq$ is defined pointwise (i.e., $(l_{11}, l_{21}) \sqsubseteq (l_{12}, l_{22})$ iff $l_{11} \sqsubseteq_1 l_{21} \wedge l_{11} \sqsubseteq_2 l_{21}$) and $\bigsqcup$ and $\sqcap$ can be computed pointwise.

Two basic domains



Creating the cross-product

Creating the cross product of the sign and even-odd lattices.



Properties of Functions

A function $f : L_1 \rightarrow L_2$ between partially ordered sets is **monotone** if:

$$\forall l, l' \in L_1 : l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')$$

The composition of monotone functions is monotone. However, being monotone does not imply being extensive ($\forall l \in L : l \sqsubseteq f(x)$). A function that maps all values to $\perp$ is clearly monotone, but not extensive.

The function f is **distributiv** if:

$$\forall l_1, l_2 \in L_1 : f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

Chains

A subset $Y \subseteq L$ of a partially ordered set $L = (L, \sqsubseteq)$ is a chain if

$$\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$$

That is, the values l_1 and l_2 are comparable; a chain is a possibly empty subset of L that is totally ordered.

The chain is finite if Y is a finite subset of L.

A sequence $(l_n)_{n \in N}$ of elements in L is an ascending chain if

$$n \leq m \Rightarrow l_n \sqsubseteq l_m$$

A descending chain is defined accordingly.

A sequence $(l_n)_n$ eventually stabilizes iff $\exists n_0 \in N : \forall n \in N : n \geq n_0 \Rightarrow l_n = l_{n_0}$

Ascending/Descending Chain Condition

- A partially ordered set L satisfies the Ascending Chain Condition if and only if all ascending chains eventually stabilize.
- A partially ordered set L satisfies the Descending Chain Condition if and only if all descending chains eventually stabilize.

(A lattice must not be finite to satisfy the ascending chain condition).

Fixed Point

- $l \in L$ is a fixed point for f if $f(l) = l$
- A least fixed point $l_1 \in L$ for f is a fixed point for f where $l_1 \sqsubseteq l_2$ for every fixed point $l_2 \in L$ for f.

Equation system

$x_1 = F_1(x_1, \ldots, x_n)$

$\vdots$

$x_n = F_2(x_1, \ldots, x_n)$

where x_i are variables and $F_i : L^n \to L$ is a collection of functions. If all functions are monotone then the system has a unique least solution which is obtained as the least-fixed point of the function $F : L^n \to L^n$ defined by:

$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$

In a lattice L with finite height, every monotone function f has a unique least fixed point given by: $fix(f) = \bigcup_{i \geq 0} f^i(\bot)$.

Data-flow analysis: Available Expressions

Determine for each program point, which expressions must have already been computed and not later modified on all paths to the program point.

The following discussion of data-flow analyses uses the more common equational approach.

Available Expressions - Example

```
           def m(initialA: Int, b: Int): Int = {
/*pc 0*/    var a = initialA

/*pc 1*/    var x = a + b;

/*pc 2*/    val y = a * b;

/*pc 3*/    while (y > a + b) {

/*pc 4*/      a = a + 1

/*pc 5*/      x = a + b

            }
/*pc 6*/    a + x
           }
```

The expression `a + b` is available every time execution reaches the test (`pc 4`).

Available Expressions - gen/kill functions

- An **expression is killed** in a block if any of the variables used in the (arithmetic) expression are modified in the block. The function $kill : Block \rightarrow \mathcal{P}(ArithExp)$ produces the set of killed arithmetic expressions.
- A **generated expression** is a non-trivial (arithmetic) expression that is evaluated in the block and where none of the variables used in the expression are later modified in the block. The function $gen : Block \rightarrow \mathcal{P}(ArithExp)$ produces the set of generated expressions.

Typically, a block is a single statement in a program's three-address code.

The underlying lattice is the powerset of all expression of the program.

If you know that a function is pure or a field is (effectively) final it is directly possible to also take these expressions into consideration.

Available Expressions - data flow equations

Let S be our program and $flow$ be a flow in the program between two statements (pc_i, pc_j).

$$AE_{entry}(pc_i) = \begin{cases} \emptyset & \text{if } i = 0 \\ \bigcap AE_{exit}(pc_h) | (pc_h, pc_i) \in flow(S) & otherwise \end{cases}$$

$$AE_{exit}(pc_i) = (AE_{entry}(pc_i) \backslash kill(block(pc_i)) \cup gen(block(pc_i)))$$

Available Expressions - Example continued I

```
/*pc 1*/  var x = a + b;
/*pc 2*/  val y = a * b;
/*pc 3*/  while (y > a + b) {
/*pc 4*/    a = a + 1
/*pc 5*/    x = a + b
          }
```

The kill/gen functions:

pc	kill	gen
1	$\emptyset$	$\{a+b\}$
2	$\emptyset$	$\{a*b\}$
3	$\emptyset$	$\{a+b\}$
4	$\{a+b, a*b, a+1\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

We get the following equations:

$AE_{entry}(pc_1) = \emptyset$
$AE_{entry}(pc_2) = AE_{exit}(pc_1)$
$AE_{entry}(pc_3) = AE_{exit}(pc_2) \cap AE_{exit}(pc_5)$
$AE_{entry}(pc_4) = AE_{exit}(pc_3)$
$AE_{entry}(pc_5) = AE_{exit}(pc_4)$
$AE_{exit}(pc_1) = AE_{entry}(pc_1) \cup \{a+b\}$
$AE_{exit}(pc_2) = AE_{entry}(pc_2) \cup \{a*b\}$
$AE_{exit}(pc_3) = AE_{entry}(pc_3) \cup \{a+b\}$
$AE_{exit}(pc_4) = AE_{entry}(pc_4) \backslash \{a+b, a*b, a+1\}$
$AE_{exit}(pc_5) = AE_{entry}(pc_5) \cup \{a+b\}$

Data-flow analysis: Naive algorithm

> to solve 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 is just a conceptual algorithm which is not to be implemented:

$x = (\perp, \ldots, \perp); do\{\ t = x; x = F(x); \}while(x \neq t);$

Data-flow analysis: Chaotic Iteration

> We exploit the fact that our lattice has the structure L^n to compute the solution $(x_1, \ldots, x_n)$:

$$x_1 = \bot; \ldots; x_n = \bot; while(\exists i : x_i \neq F_i(x_1, \ldots, x_n))\{\, x_i = F_i(x_1, \ldots, x_n); \}$$

Available Expressions - Example continued II

```
/*pc 1*/  var x = a + b;
/*pc 2*/  val y = a * b;
/*pc 3*/  while (y > a + b) {
/*pc 4*/    a = a + 1
/*pc 5*/    x = a + b
          }
```

Solution:

pc	kill	gen
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

References