50.054 Sign Analysis and Lattice Theory

ISTD, SUTD

Learning Outcomes

1. Explain the objective of Sign Analysis
2. Define Lattice and Complete Lattice
3. Define Monotonic Functions
4. Explain the fixed point theorem
5. Apply the fixed point theorem to solve equation constraints of sign analysis

Recall the following example

```
// SIMP1
x = input;
while (x >= 0) {
    x = x - 1;
}
y = Math.sqrt(x);
// error, can't apply
// sqrt() to a negative number.
return y;
```

```
// PA1
1: x <- input
2: t1 <- 0 < x
3: ifn t1 goto 6
4: x = x - 1
5: goto 2
6: y <-  Math.sqrt(x)
// x is definitely negative
7: rret <- y
8: ret
```

Sign Analysis

To statically determine the possible signs of integer variables at the end of a statement in a program.

What we hope to see is.

```
// SIMP1
x = input;          // x could be +, - or 0
while (x >= 0) {    // x could be +, - or 0
    x = x - 1;      // x could be +, - or 0
}                   // x must be -
y = Math.sqrt(x); // x must be -, y could be +, - or 0
return y;           // x must be -, y could be +, - or 0
```

Sign Analysis as Type Inference?

```
// SIMP1
x = input;          // (alpha_x, alpha_input)
while (x >= 0) {    // (alpha_x, int)
    x = x - 1;      // (alpha_x, int)
}                   //
y = Math.sqrt(x);   // (alpha_x, ???)
return y;           //
```

Suppose we have types `int`, `posint`, `negint` and `zero`, though `???` must be `negint`, we can't infer it.

- Type Inference is flow-insensitive
  - We can give only 1 type per type variable throughout the entire program
- Type Inference is path-insensitive
  - We can't distinguish the paths
    - in the while loop.
    - after the whlie loop.

Sign Analysis as Type Inference?

```
// SIMP2
x = 0;          // (alpha_x, zero)
x = x + 1;      // (alpha_x, ???)
return x;
```

Suppose we have types `int`, `posint`, `negint` and `zero`, though `???` must be `posint`, we can't infer it.

- Type Inference is flow-insensitive
  - We can give only 1 type per type variable throughout the entire program
- Type Inference is path-insensitive
  - We can't distinguish the paths
    - in the while loop.
    - after the whlie loop.

Sign Analysis as Abstract Domain Equation

Abstract values

- $+$ all the positive integers
- $-$ all the negative integers
- 0 the zero.
- $\top$ can be positive, negative, and zero
- $\bot$ no information, undefined.

Sign Analysis as Abstract Domain Equation

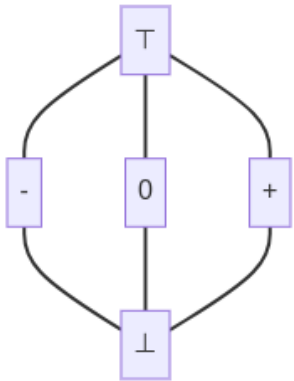
Abstract values

- $+$ the set of all the positive integers
- $-$ the set of all the negative integers
- 0 the set $\{0\}$.
- $\top$ the set of all integers.
- $\bot$ the empty set $\{\}$.
- $\bot \subseteq +$, $\bot \subseteq -$ and $\bot \subseteq 0$
- $+ \subseteq \top$, $0 \subseteq \top$ and $- \subseteq \top$

Sign Analysis as Abstract Domain Equation

Abstract values

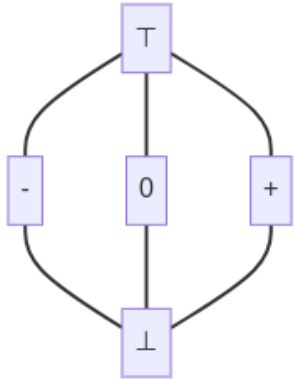
- $+$ the set of all the positive integers
- $-$ the set of all the negative integers
- 0 the set $\{0\}$.
- $\top$ the set of all integers.
- $\bot$ the empty set $\{\}$.
- $\bot \subseteq +$, $\bot \subseteq -$ and $\bot \subseteq 0$
- $+ \subseteq \top$, $0 \subseteq \top$ and $- \subseteq \top$



Sign Analysis as Abstract Domain Equation

```
// SIMP2
x = 0;              // {(x, 0)}
x = x + 1;          // {(x, 0 ++ +)}
return x;
```

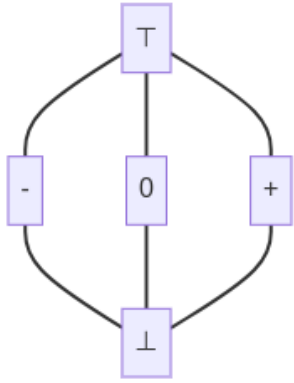
++	⊤	+	-	0	⊥
⊤	⊤	⊤	⊤	⊤	⊥
+	⊤	+	⊤	+	⊥
-	⊤	⊤	-	-	⊥
0	⊤	+	-	0	⊥
⊥	⊥	⊥	⊥	⊥	⊥



Sign Analysis as Abstract Domain Equation

```
// SIMP1
x = input;          // {(x,top), (input, top)}
while (x >= 0) {    // {(x,top), (input, top)}
    x = x - 1;      // {(x,top), (input, top)}
}                   //
y = Math.sqrt(x); // {(x,top), (input, top)}
return y;           //
```

–	⊤	+	-	0	⊥
⊤	⊤	⊤	⊤	⊤	⊥
+	⊤	⊤	+	+	⊥
-	⊤	-	⊤	-	⊥
0	⊤	-	+	0	⊥
⊥	⊥	⊥	⊥	⊥	⊥



The analysis is still path-insensitive.

- We will come back to this issue.

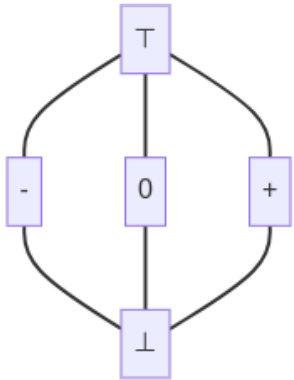
Partial Order

S is a *partial order* iff there exists a binary relation $\sqsubseteq$

1. reflexivity: $\forall x \in S, x \sqsubseteq x$
2. transitivity: $\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z$ implies $x \sqsubseteq z$.
3. anti-symmetry: $\forall x, y \in S, x \sqsubseteq y \wedge y \sqsubseteq x$ implies $x = y$.

For example,

- The set of abstract values $\{\top, \bot, +, -, 0\}$ is a partial order, if we let $\sqsubseteq = \subseteq$



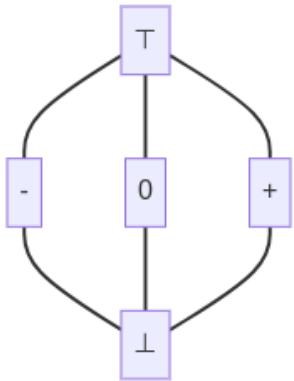
Upper Bound and Least Upper Bound

S is a partial order and $T \subseteq S$, $y \in S$.

- y is an upper bound of T (written as $T \sqsubseteq y$) iff $\forall x \in T, x \sqsubseteq y$.
- y is the least upper bound of T (written as $y = \bigsqcup T$) iff $\forall z \in S, T \sqsubseteq z$ implies $y \sqsubseteq z$.

For example

- The least upper bound of abstract values $\{\top, \bot, +, -, 0\}$ is $\top$,
- What is the least upper bound of $\{\bot, +, 0\}$?
- What is the least upper bound of $\{\bot, 0\}$?



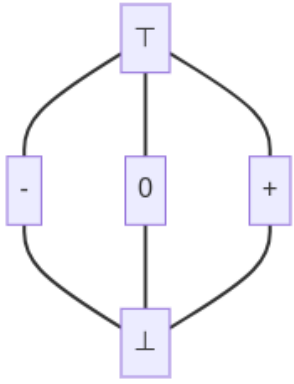
Lower Bound and Greatest Lower Bound

S is a partial order and $T \subseteq S$, $y \in S$.

- y is a lower bound of T (written as $y \sqsubseteq T$) iff $\forall x \in T, y \sqsubseteq x$.
- y is the greatest lower bound of T (written as $y = \sqcap T$) iff $\forall z \in S, z \sqsubseteq T$ implies $z \sqsubseteq y$.

For example

- The greatest lower bound of abstract values $\{\top, \bot, +, -, 0\}$ is $\bot$,
- What is the greatest lower bound of $\{\top, +, 0\}$?
- What is the greatest lower bound of $\{\top, 0\}$?



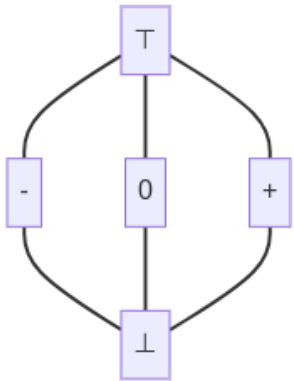
Meet and Join

Let S be a partial order, and $x, y \in S$.

1. We define the join of x and y as $x \sqcup y = \bigsqcup\{x, y\}$.
2. We define the meet of x and y as $x \sqcap y = \sqcap\{x, y\}$.

For example

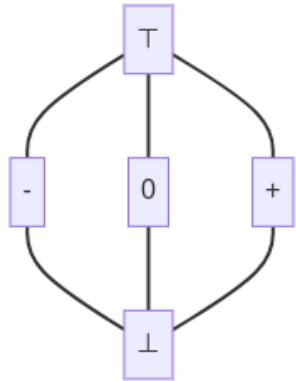
- $\bot \sqcup + = +$
- $0 \sqcup + = \top$
- $0 \sqcap + = \bot$
- $0 \sqcap \top = 0$



Lattice and Complete Lattice

A partial order $(S, \sqsubseteq)$ is a *lattice* iff $\forall x, y \in S$, $x \sqcup y$ and $x \sqcap y$ exist.
A partial order $(S, \sqsubseteq)$ is a *complete lattice* iff $\forall X \subseteq S$, $\bigsqcup X$ and $\sqcap X$ exist.

- $(\{\top, \bot, +, -, 0\}, \sqsubseteq)$ is a lattice, also a complete lattice.
- $(\mathbb{R}, \leq)$ is a lattice but not a complete lattice.
- Lemma
  - Let S be a non empty finite set and $(S, \sqsubseteq)$ is a lattice, then $(S, \sqsubseteq)$ is a complete lattice.



Some commonly use lattice for program analysis

- Powerset Lattice
- Product Lattice
- Map Lattice

Powerset Lattice

Let A be a set.

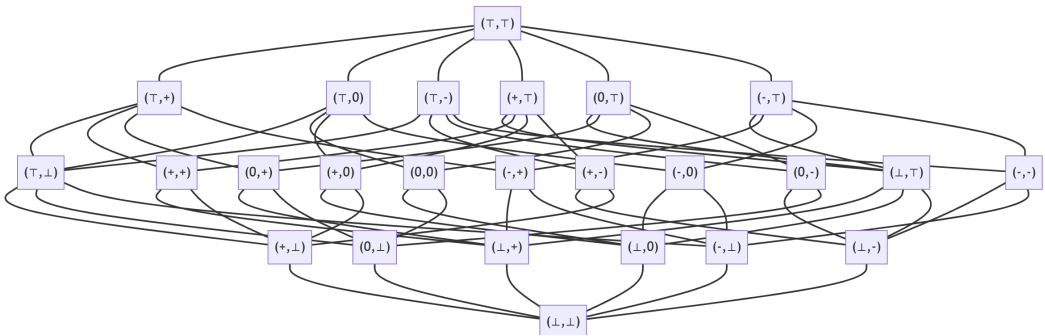
- $\mathcal{P}(A)$ denotes the powerset of A.
- $(\mathcal{P}(A), \subseteq)$ forms a complete lattice.
- What is the top element?
- What is the bottom element?
- We will show it in jamboard $A = \{x, y, z\}$
- We will need powerset lattice in Liveness Analysis

Product Lattice

Let $L_1, ..., L_n$ be complete lattices, then $(L_1 \times ... \times L_n)$ is a complete lattice where the $\sqsubseteq$ is defined as

$$(x_1, ..., x_n) \sqsubseteq (y_1, ..., y_n) \texttt{ iff } \forall i \in [1, n], x_i \sqsubseteq y_i$$

- L^n as a short-hand for $(L_1 \times ... \times L_n)$. For example $\mathit{VarSign} \times \mathit{VarSign}$.



Map Lattice

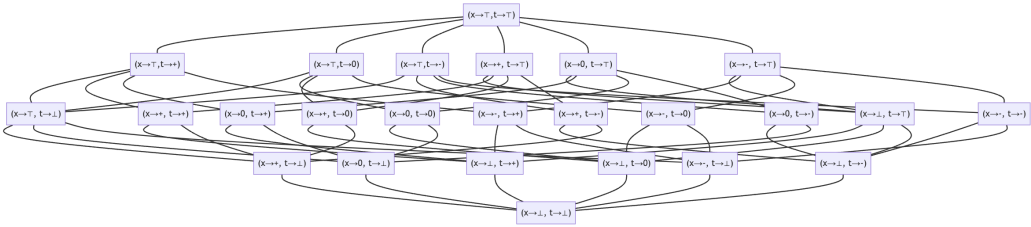
Let L be a complete lattice, A be a set. Let $A \to L$ denotes a set of functions

$$\{m \mid x \in A \wedge m(x) \in L\}$$

and the $\sqsubseteq$ relation among functions $m_1, m_2 \in A \to L$ is defined as

$$m_1 \sqsubseteq m_2 \texttt{ iff } \forall x \in A, m_1(x) \sqsubseteq m_2(x)$$

Then $A \to L$ is a complete lattice. We can view $A \to L$ like a Scala Map `Map[A,L]`. For example $Var \mapsto VarSign$, where $Var = \{x, t\}$



Formalizing Sign Analysis with Lattice

```
// PA2          // s0 = [x -> top]
1: x <- 0       // s1 = s0[x -> 0]
2: x = x + 1    // s2 = s1[x -> s1(x) ++ +]
3: rret <- x    // s3 = s2
4: ret
```

- We apply the analysis on PA instead of SIMP.
- `s0`, `s1`, `s2` are `s3` elements of the map lattice $\mathit{Var} \mapsto \mathit{VarSign}$.
- We call each of the element abstract state
- `s(x)` returns the associated abstract value of `x` in abstract state `s`
- `s[x -> +]` returns a new abstract state same as `s` except `x` is mapped to +, (i.e. update function in FP style.)

Solving the equation system

- In the absence of loops, we can process the equations from top to bottom.

```
s0 = [x -> top]
s1 = s0[x -> 0]
s2 = s1[x -> s1(x) ++ +]
s3 = s2
```

we have a solution

```
s0 = [x -> top]
s1 = [x -> 0]
s2 = [x -> +]
s3 = [x -> +]
```

Solving the equation system

- In the presence of loops, we can't just process them from top to bottom

```
// PA4                  // s0 = [x -> top, y -> top, t -> top]
1: x <- input           // s1 = s0
2: y <- 0               // s2 = s1[y -> 0]
3: t <- x > 0           // s3 = upperbound(s2,s7)[t -> top]
4: ifn t goto 8         // s4 = s3
5: y <- y + 1           // s5 = s4[y -> s4(y) ++ +]
6: x <- x - 1           // s6 = s5[x -> s5(x) -- +]
7: goto 3               // s7 = s6
8: rret <- y            // s8 = s4
9: ret
```

Monotonic Function

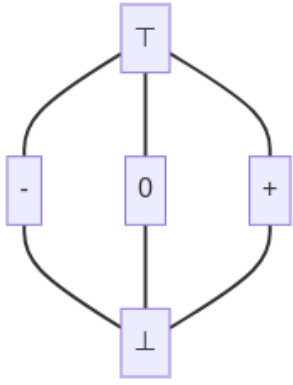
Let L_1 and L_2 be lattices, a function $f : L_1 \longrightarrow L_2$ is *monotonic* iff $\forall x, y \in L_1, x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.
This function f can be viewed as Scala functoins.
Example

$$f_1(x) \quad = \quad \top$$

Function f_1 is monotonic because

$$\begin{aligned} f_1(\bot) &= \top \\ f_1(0) &= \top \\ f_1(+) &= \top \\ f_1(-) &= \top \\ f_1(\top) &= \top \end{aligned}$$

and $\top \sqsubseteq \top$



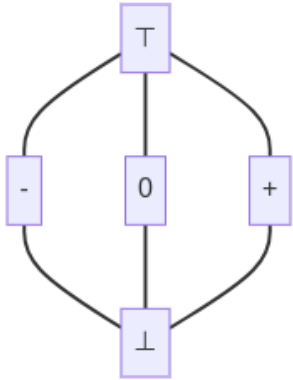
Monotonic Function

Let L_1 and L_2 be lattices, a function $f : L_1 \longrightarrow L_2$ is *monotonic* iff $\forall x, y \in L_1, x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.
Example

$$f_2(x) \quad = \quad x \sqcup +$$

is f_2 monotonic? Recall $\sqcup$ computes the least upper bound of the operands

$$\begin{aligned} f_2(\bot) &= \bot \sqcup + = + \\ f_2(0) &= 0 \sqcup + = \top \\ f_2(+) &= + \sqcup + = + \\ f_2(-) &= - \sqcup + = \top \\ f_2(\top) &= \top \sqcup + = \top \end{aligned}$$



Commonly used Monotonic Functions

- Every constant function f is monotonic.
- $\sqcup$ is a function $L \times L \to L$, then $\sqcup$ is monotonic.
- $\sqcap$ is a function $L \times L \to L$, then $\sqcap$ is monotonic.
- Monotonic function returning a map lattice composed with map update with another monotonic function is monotonic.
  - Let $f : L_1 \to (A \to L_2)$ be a monotonic function from a lattice L_1 to a map lattice $A \to L_2$.
  - Let $g : L_1 \to L_2$ be another monotonic function.
  - Then $h(x) = f(x)[a \mapsto g(x)]$ is a monotonic function of $L_1 \to (A \to L_2)$.

```
val a:A = ... // a is an element of A, where A is a ground type.
def f[L1,L2](x:L1):Map[A,L2] = ... // f is monotonic.
def h[L1,L2](x:L1):Map[A,L2] = f(x) + (a -> g(x))
// h is also monotonic.
```

Fixed Point Theorem

Let L be a complete lattice with finite height, every monotonic function f has a unique least fixed point point, namely $\texttt{lfp}(f)$, defined as

$$\texttt{lfp}(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

Where $f^n(x)$ is a short hand for

$$\overbrace{f(...(f(x)))}^{n \text{ times}}$$

The height of a complete lattice is the length of the longest path from $\top$ to $\bot$.

Intuitively speaking, the theorem says that if we keep applying a monotonic function to an element in a lattice, the result will eventually be bounded.

Naive Fixed Point Algorithm

input: a function `f`.

1. initialize `x` as the least element of the entire lattice.
2. apply `f(x)` as `x1`
3. check `x1 == x`
   3.1 if true return `x`
   3.2 else, update `x = x1`, go back to step 2.

For instance, if we apply the above algorithm to the f_2 with the sign lattice we have the following iterations. Let x be the abstract state.

1. $x = \bot, x_1 = f_2(x) = +$
2. $x = +, x_1 = f_2(x) = +$
3. fixed point is reached, return x.

What is the time complexity of the naive fixed point algorithm?

Applying Naive Fixed Point Algorithm to the sign analysis problem

Objective: to turn the equation system into a monotonic function.

```
s0 = [x -> top]
s1 = s0[x -> 0]
s2 = s1[x -> s1(x) ++ +]
s3 = s2
```

$f : (Var \mapsto VarSign)^4 \rightarrow (Var \mapsto VarSign)^4$

```
f((s0,s1,s2,s3)) = (
    [x -> top],
    s0[x -> 0],
    s1[x -> s1(x) ++ +]
    s2
)
```

Note that "Monotonic function returning a map lattice composed with map update with another monotonic function is monotonic", The abstract function ++ is a monotonic function.

Applying Naive Fixed Point Algorithm to the sign analysis problem

```
s0 = [x -> top, y -> top, t -> top]
s1 = s0
s2 = s1[y -> 0]
s3 = upperbound(s2,s7)[t -> top]
s4 = s3
s5 = s4[y -> s4(y) ++ +]
s6 = s5[x -> s5(x) -- +]
s7 = s6
s8 = s4
```

```
f((s0,s1,s2,s3,s4,s5,s6,s7,s8)) = (
    [x -> top, y -> top, t -> top],
    s0,
    s1[y -> 0],
    join(s2,s7)[t -> top],
    s3,
    s4[y -> s4(y) ++ +],
    s5[x -> s5(x) -- +],
    s6,
    s4
)

join(s1,s2) = upperbound(s1,s2) // the
```

Constructing the monotonic function for SIMP in general

1. Define the abstract domain, i.e. the abstract states.
2. Show that the abstract domain a complete lattice.
3. Define the *join* function among abstract states. For sign analysis

$$join(s) = \bigsqcup pred(s)$$

4. Define the monotonic function by cases. For sign analysis

- case $l == 0$, $s_0 = [x \mapsto \top \mid x \in V]$
- case $l : t \leftarrow src$, $s_l = join(s_l)[t \mapsto join(s_l)(src)]$
- case $l : t \leftarrow src_1\ op\ src_2$, $s_l = join(s_l)[t \mapsto (join(s_l)(src_1)\ abs(op)\ join(s_l)(src_2))]$
- other cases: $s_l = join(s_l)$

V denotes all variables in the PA program.

Constructing the monotonic function for SIMP in general

$abs(op)$ maps a SIMP operator into a abstract function.

$$
\begin{aligned}
abs(+) &= ++ \\
abs(-) &= -- \\
abs(*) &= ** \\
abs(<) &= << \\
abs(==) &= ===
\end{aligned}
$$

Let S be a set of abstract states $join(S)$ returns a map lattice object m. $join(S)(src)$ extract the sign abstract values from the SIMP operand src.

$$
m(c) = \begin{cases} 0 & c == 0 \\ + & c > 0 \\ - & c < 0 \end{cases}
$$

$$
m(t) = \begin{cases} v & t \mapsto v \in m \\ error & otherwise \end{cases}
$$

$$
m(r) = error
$$