# Static Program Analysis

Xiangyu Zhang

# What is static analysis



- Static analysis analyzes a program without executing it.
- Static analysis is widely used in bug finding, vulnerability detection, property checking
  - Easier to apply compared to dynamic analysis (as long as you have code)
    - The user does not even need to know how to run it
  - Better scalability compared to some dynamic analysis (e.g. tracing)
  - Findbug, coverity, codesurfer

# Two kinds of Static Analysis

- Syntax/structure oriented analysis
  - They don't try to understand the semantics of a program. Instead, they look at syntax and structure of a program
    - CFG, dominator, post-dominator, loop detection
  - A lot of applications
    - Code clone detection (text comparison, AST comparison, CFG comparison)
    - Malware analysis
    - Serve as the foundation for other advanced static/dynamic analysis
  - Limitation: cannot reason about program semantics and program state
- Semantics oriented analysis (our focus)

# Lets start with the Simplest Static Analysis

- What are the possible definitions for each use

```
z=…
x=…
if (…)
    x=…
else
    s1
z=…
if (…)
    y=…x…
else
    y=…z…
```

- What are the possible call targets

```
p=F1   /*F1, F2, F3, F4, F5 are functions*/
q=F2
x=input ()
if (…)
   q=F3
else
   p=F4
if (…)
   p=F5
else
   p=q;
(*p) (…)
```

- What is the range of possible values for a integer var.

```
x=10
y=input()
i=x+y
if (i>20)
   i=20
else
   z=input()
   if (3<z<5)
      i=i-z
print z
```

# The first ingredient of static analysis

- Abstract domain
  - The results we want to compute by static analysis
- Transfer function
  - How the abstract values are computed/updated at each relevant instruction
    - Need to consider the instruction semantics

- What are the possible call targets

```
x=F1        /*F1, F2, and F3 are functions*/
y=F2
q=&x
if (…)
   x=F3
else
   p=&x
if (…)
   p=q
else
   p=&y;
*(*p) (…)
```

# What about loops

- When shall we terminate a loop path?
  - Analyze the possible sign of a variable

```
x=input()
while (…)
    x=-x
```

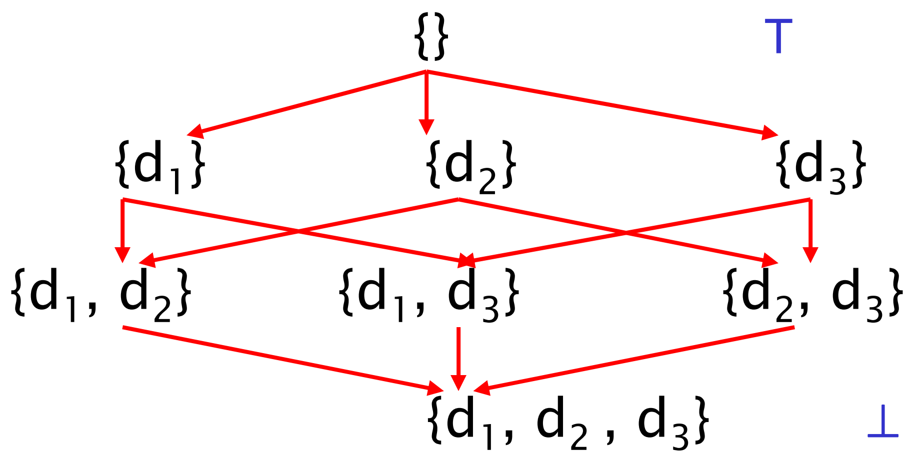
- Since we are always interested in the aggregation of abstract values along all paths. If the aggregation stabilizes, we shall terminate
  - Monotonically growth
  - The abstract domain is finite

# Semi-lattice

- A <u>semi-lattice</u> is a domain of values $V$ and a meet operator $\wedge$ such that,
  - $\forall$ $a$, $b$, & $c$ $\in V$ :
    1. $a \wedge a = a$ (idempotent)
    2. $a \wedge b = b \wedge a$ (commutative)
    3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ (associative)
  - $\wedge$ imposes a partial order on $V$, $\forall$ $a$, $b$, & $c$ $\in V$ :
    1. $a \geq b \Leftrightarrow a \wedge b = b$
    2. $a > b \Leftrightarrow a \geq b$ and $a \neq b$
    3. $a \geq b$ and $b \geq c$, then $a \geq c$
  - A semi-lattice has a <u>top</u> element, denoted T
    1. $\forall a \in V, a \leq T$
    2. $\forall a \in V, T \wedge a = a$

# Semi-lattices for previous examples

- Def[x@n]: the possible definitions of x at n



- Lattice + monotonicity + finite height = termination
- Are we there yet?
  - Path explosion, e.g. a program with n diamonds.

# Avoid Analyzing Individual Paths



- Analyze multiple paths at a time and compute aggregate information directly.
  - $Def_{in}[x@n]$: all the possible definitions of x along some path reaching n (before getting through n)
    $Def_{in}[x@n] = \wedge_{n's\ predecessor\ n_p} Def_{out}[x@n]$
  - For any x!=y (node n is "y=…")
    $Def_{out}[x@n] = Def_{in}[x@n]$
  - $Def_{out}[y@n] = \{n\}$

# Other Examples

- Call target analysis
- Range analysis

# Worklist Algorithm

For each block node n and every variable x
$AD^{in}[x@n]=Ad_{out}[x@n] = Ø$
change = true;
while change do begin
    change = false;
    for any n and x
        $AD^{in}[x@n]=\wedge_{n's\ predecessor\ n_p} AD_{out}[x@n]$
        oldvalue = $Ad_{out}[x@n]$ ;
        $Ad_{out}[x@n] = F(AD^{in}[x@n])$
        if $Ad_{out}[x@n]$ != oldvalue then change = true;
    end
end

# Example for Computing Dependences

```
Input (x,y);
if (x<0)
   p=-y;
else
   p=y;
z=1
while (p!=0)
   z=z*x
   p=p-1;
Output(z);
```

# Lost of Precision by Directly Computing Aggregate Information

```
x=foo();
y=gee();
if (…)
   p=&x;
   q=&x;
else
   p=&y;
   q=&y;
*p=*q
*(*p)();
```

```
if (…)
   a=1;
   b=2;
else
   a=2;
   b=1;
c=a+b
```

- Distributive analysis: the aggregation of individual path analysis results is equivalent to computing the aggregate information directly

$$F(a \wedge b)=F(a) \wedge F(b)$$

# Analyzing Model Output Value Range

```
for (L=1 to 2) {
   for (i=0 to 1) {
      a[L,i]=0;
      for (j=0 to 1) {
         a[L,i]=a[L,i]+w[L,i,j]*a[L-1,j];
      }
      a[L,i]=a[L,i]+b[i];
      if (a[L,i]<0)
         a[L,i]=0;
    }
}
```

```
w[0]= { {2,4}, {1,3}}   w[1]={ {3,2}, {7,1}}
b[0]=10                 b[1]=-20
```

# Alias Analysis

- For each pointer variable, determine the set of global variables and the heap objects that may be pointed-to by the variable
  - One of the most important analyses

# Example

```
1:    p=&x
2:    q=malloc(10)
3:    *q=p
4:    t=q+2
5:    *t=malloc(5)
6:    r=(*t)
7    (*r)=&p
```

# A More Efficient Alias Analysis

- Traversing different paths is very expensive
- How about we ignore the control flow
  - Eliminate strong update
    - x =…     never overwrites the PointsTo set of x, but rather add to it

```
x=&y
x=&z
t=x
```

# Analysis Rules



$$\frac{x = \&y}{x \rightarrow y}$$

$$\frac{x = {}^{L}malloc(y)}{x \rightarrow L}$$

$$\frac{x = y \quad y \rightarrow t}{x \rightarrow t}$$

$$\frac{(*\,x) = y \quad y \rightarrow t \quad x \rightarrow p}{p \rightarrow t}$$

$$\frac{x = y + z \quad y \rightarrow t}{x \rightarrow t}$$

$$\frac{x = (*\,y) \quad y \rightarrow t \quad t \rightarrow q}{x \rightarrow q}$$

# Example

```
p=malloc (10)
(*p)=&x
q=p+1
(*q)=&y
r=q+1
(*r)= &z
i=0;
t=p;
while  (i<3) {
    z=(*t);
    t=t+2;
    i=i+1;
}
x=(*z);
```

# Flow Sensitive and Flow Insensitive Analysis



- With and without respecting control flow
- The analyses we have learned, except the preceding alias analysis, are flow sensitive
- Other flow insensitive analysis
  - Type inference

# Summary

- The essence of static analysis is similar to dynamic analysis
  - Execute it without the concrete values (abstract interpretation)
  - We can express our analysis with abstract semantics just like concrete semantics
    - You can implement static analysis just like a dynamic analysis
  - Two important properties: termination and soundness
- For better scalability, we come up with different approximations
  - Merge-before-continue semantics
  - Flow insensitive analysis