Static Analysis

Dataflow Analysis

Roadmap

- **Overview.**
- Four Analysis Examples.
- Analysis Framework – Soot.
- Theoretical Abstraction of Dataflow Analysis.
- Inter-procedure Analysis.
- Taint Analysis.

Overview

- Static analysis is a program analysis technique performed without actually executing programs.
- Data flow analysis is a process of deriving information about the run time behavior of a program.
- Usage: compiler, IDE and security.

SSA

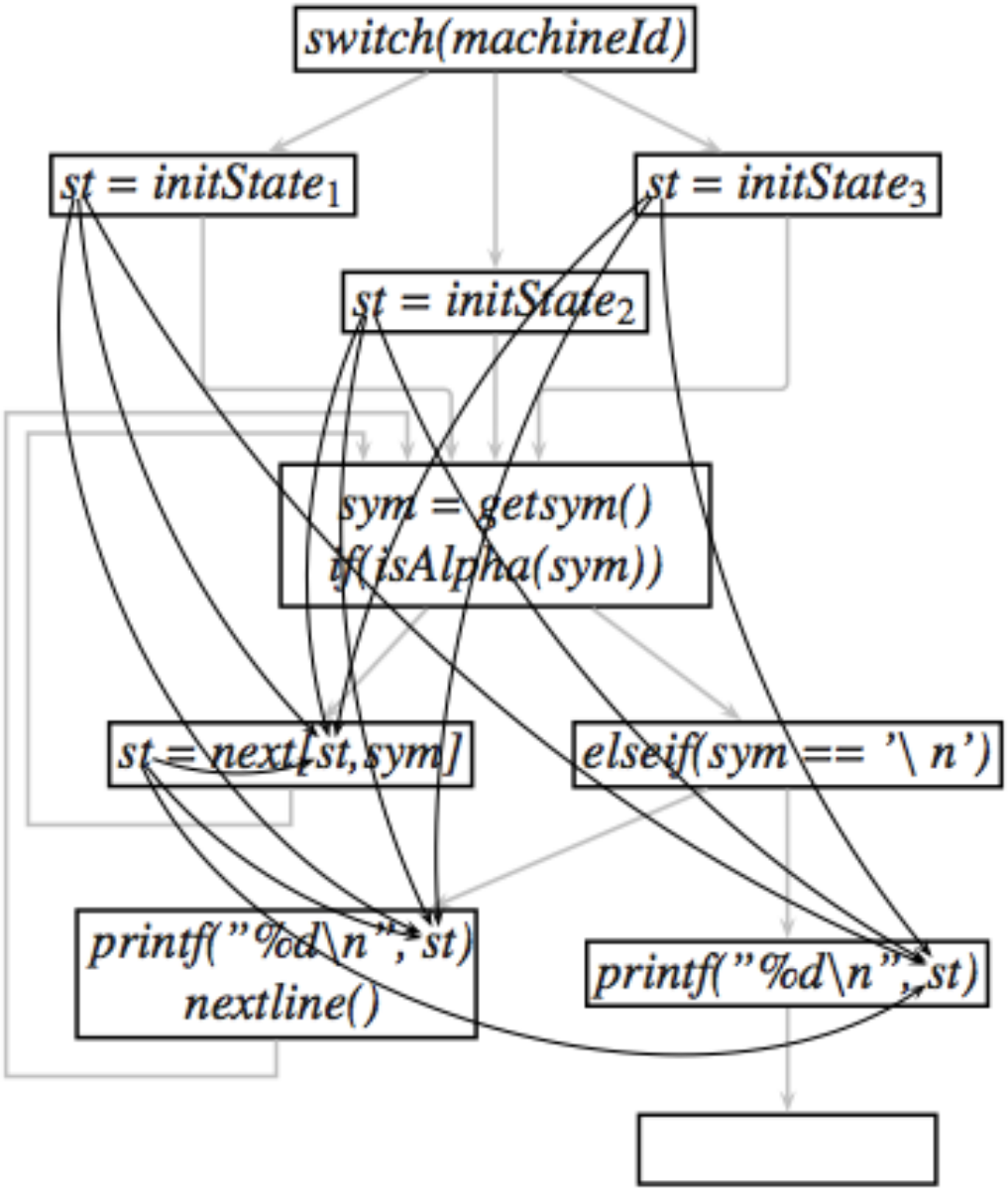
- Requires that each variable is assigned exactly once.

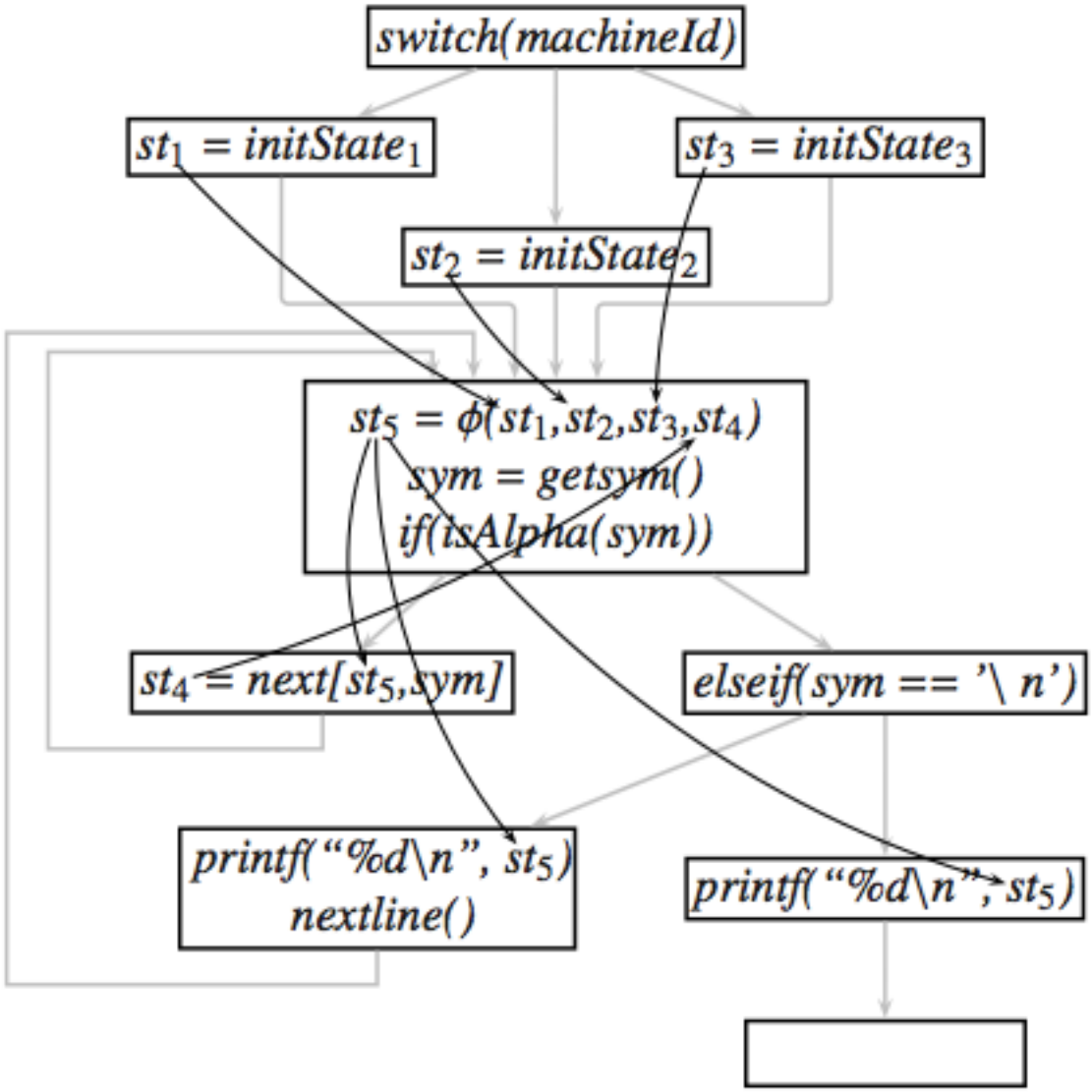
```
y := 1
y := 2
x := y
```

```
y1 := 1
y2 := 2
x1 := y2
```

- Def-use chain:
  - Def-use chains are used to propagate data flow information.
  - The analysis algorithm takes time proportional to the product of the total number of def-use edges
- Benefits:
  - Data flow analysis could be easier and faster.
  - Reduce the number of def-use chains. (m*n vs m+n)

```
switch(machineId)
{  case1:
     st = initState1;
     break;
   case2:
     st = initState2;
     break;
   case3:
     st = initState3;
}
while (1)
{  sym = getsym();
   if(isAlpha(sym))
     st = next[st,sym];
   elseif(sym == '\n')
   {  printf("%d\n", st);
     nextline();
   }
   else
   {  printf("%d\n", st);
     break;
   }
}
```

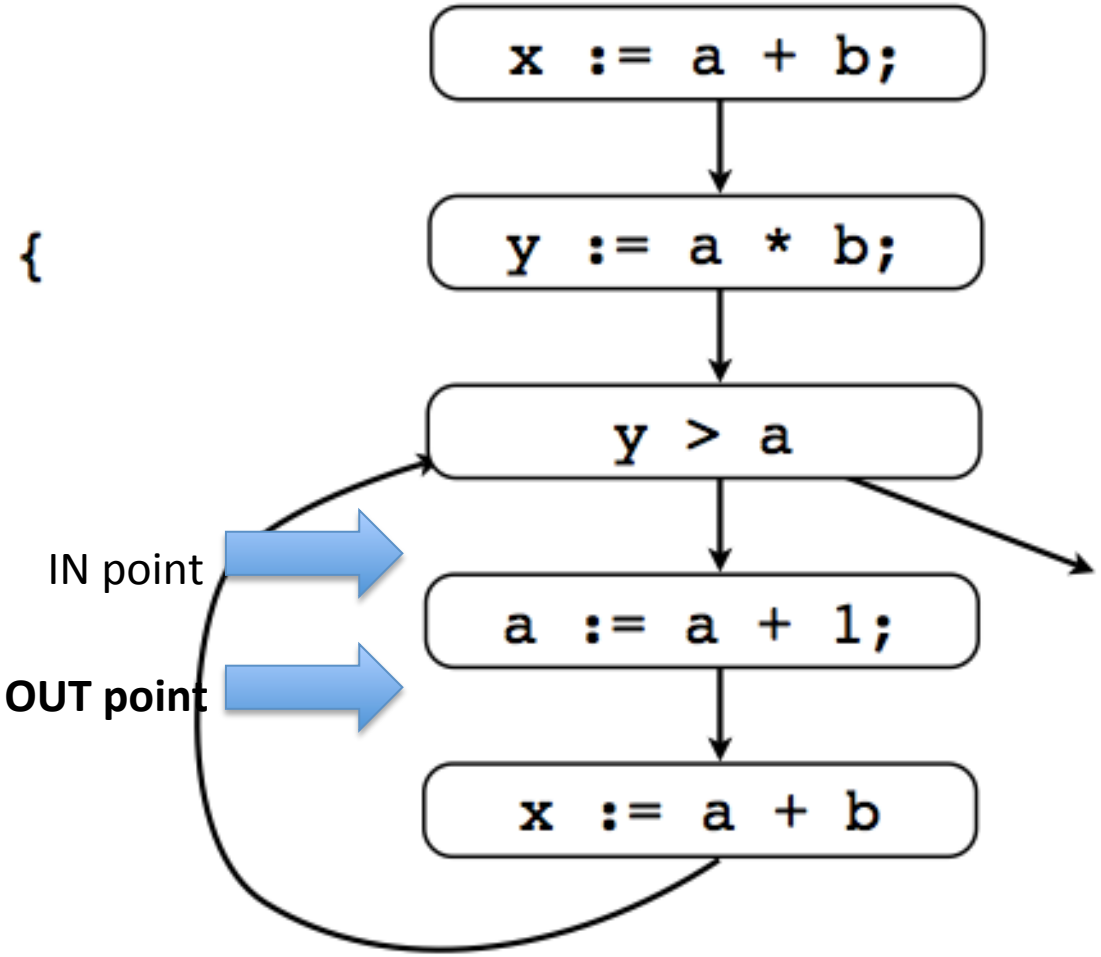



switch(machineId)
st1 = initState1
st3 = initState3
st2 = initState2
st5 = φ(st1,st2,st3,st4)
sym = getsym()
if(isAlpha(sym))
st4 = next[st5,sym]
elseif(sym == '\ n')
printf("%d\n", st5)
nextline()
printf("%d\n", st5)

Control Flow Graph (CFG)

- A control flow graph is a representation of a program that makes certain analyses (including dataflow analyses) easier.
- Usually built on Intermediate representation:
  - Single static assignment (SSA) form.
- Statements may be
  - Assignments: x := y or x := y op z or x := op y
  - Branches: goto L or if b then goto L
- A directed graph where
  - Each node represents a statement
  - Edges represent control flow

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



Roadmap

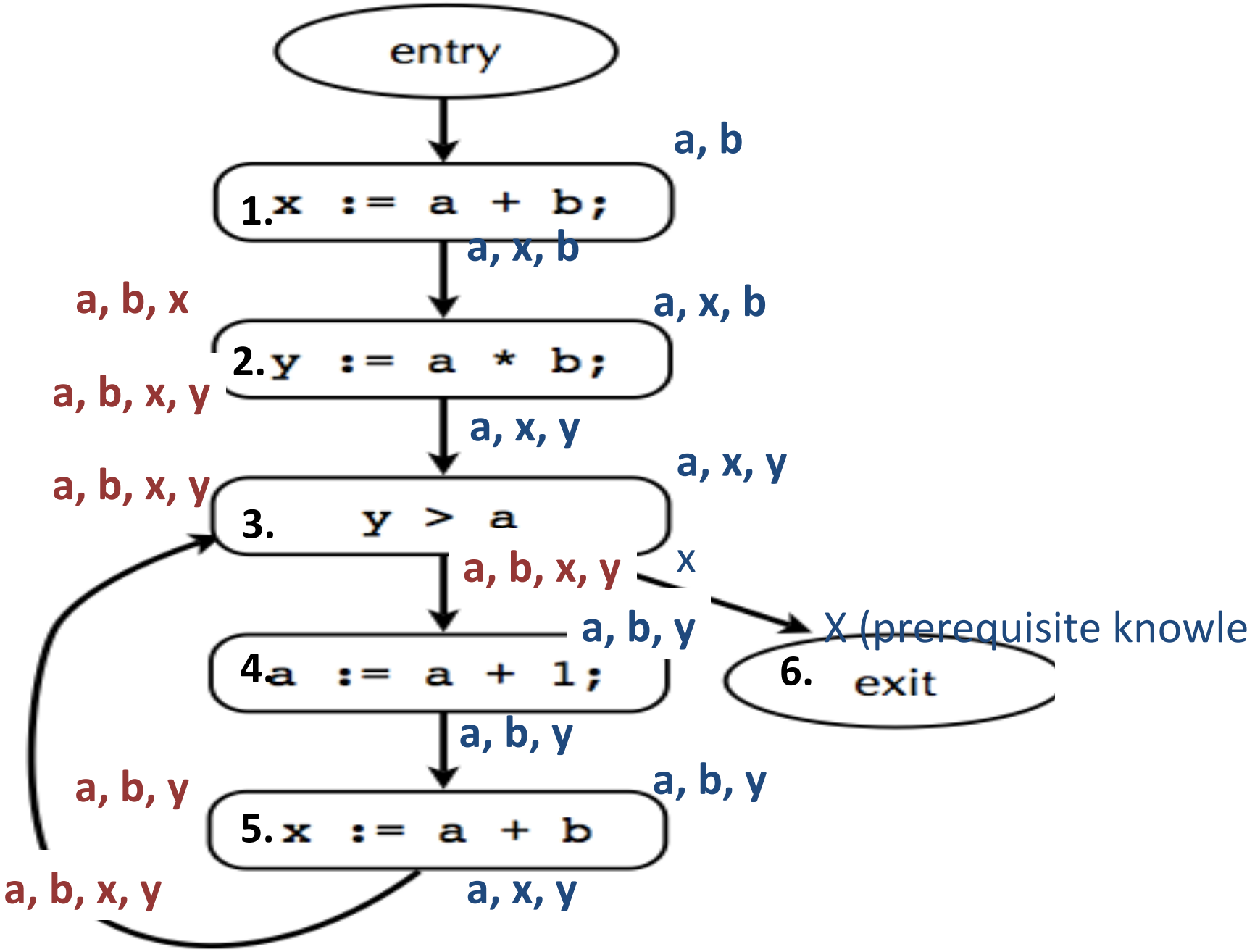
- Overview.
- **Four Analysis Examples**.
- Analysis Framework – Soot.
- Theoretical Abstraction of Dataflow Analysis.
- Inter-procedure Analysis.
- Taint Analysis.

Example

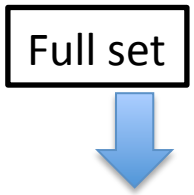
- Available expressions
- Reaching definitions
- Live variables
- Very busy expressions

- Data-flow-analysis-example.pdf

Blackboard


entry
a, b
1. x := a + b;
a, x, b
a, b, x
a, x, b
2. y := a * b;
a, b, x, y
a, x, y
a, b, x, y
a, x, y
3. y > a
a, b, x, y
X
a, b, y
X (prerequisite knowle
4 a := a + 1;
6. exit
a, b, y
a, b, y
a, b, y
5. x := a + b
a, b, x, y
a, x, y

Forward Must Dataflow Algorithm



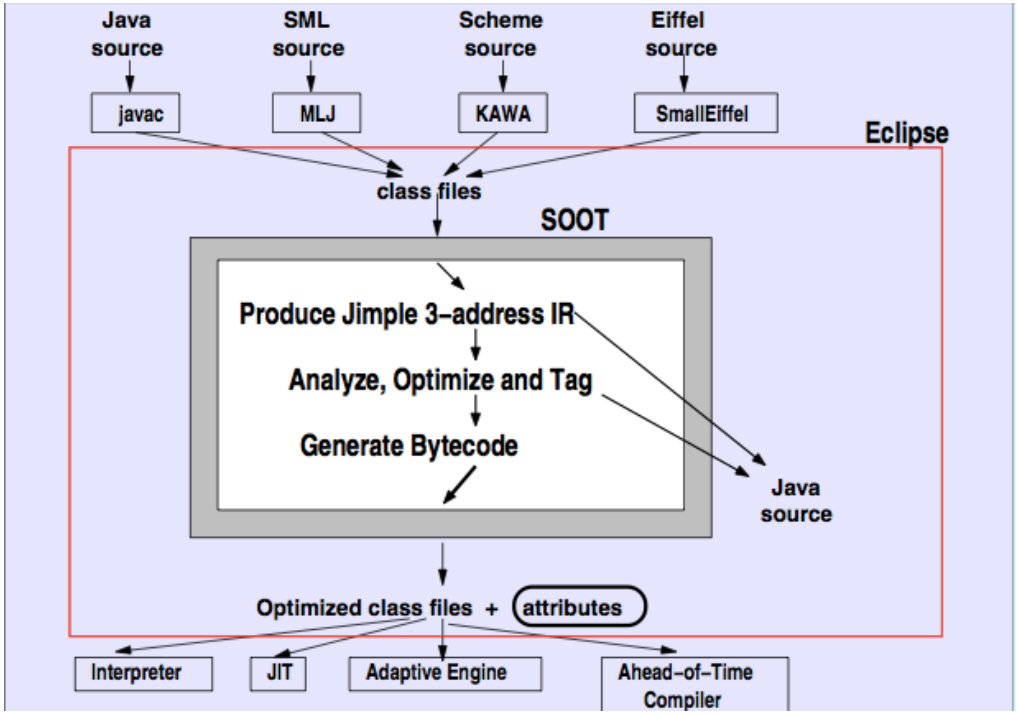

```
1  Out(s) = ⊤ for all statements s W := { all statements }
2  repeat {
3      Take s from W
4      In(s) := ∩s′ ∈ pred(s) Out(s′)
5      temp := Gen(s) ∪ (In(s) - Kill(s))
6      if (temp != Out(s)) {
7          Out(s) := temp
8          W := W ∪ succ(s)
9      }
10 } until W = ∅
```

Roadmap

- Overview.
- Four Analysis Examples.
- **Analysis Framework – Soot.**
- Theoretical Abstraction of Dataflow Analysis.
- Inter-procedure Analysis.
- Taint Analysis.

Use Framework to Implement Analysis

- Soot is a framework to analyze and optimize Java or Android programs.



Four Steps to Use Soot

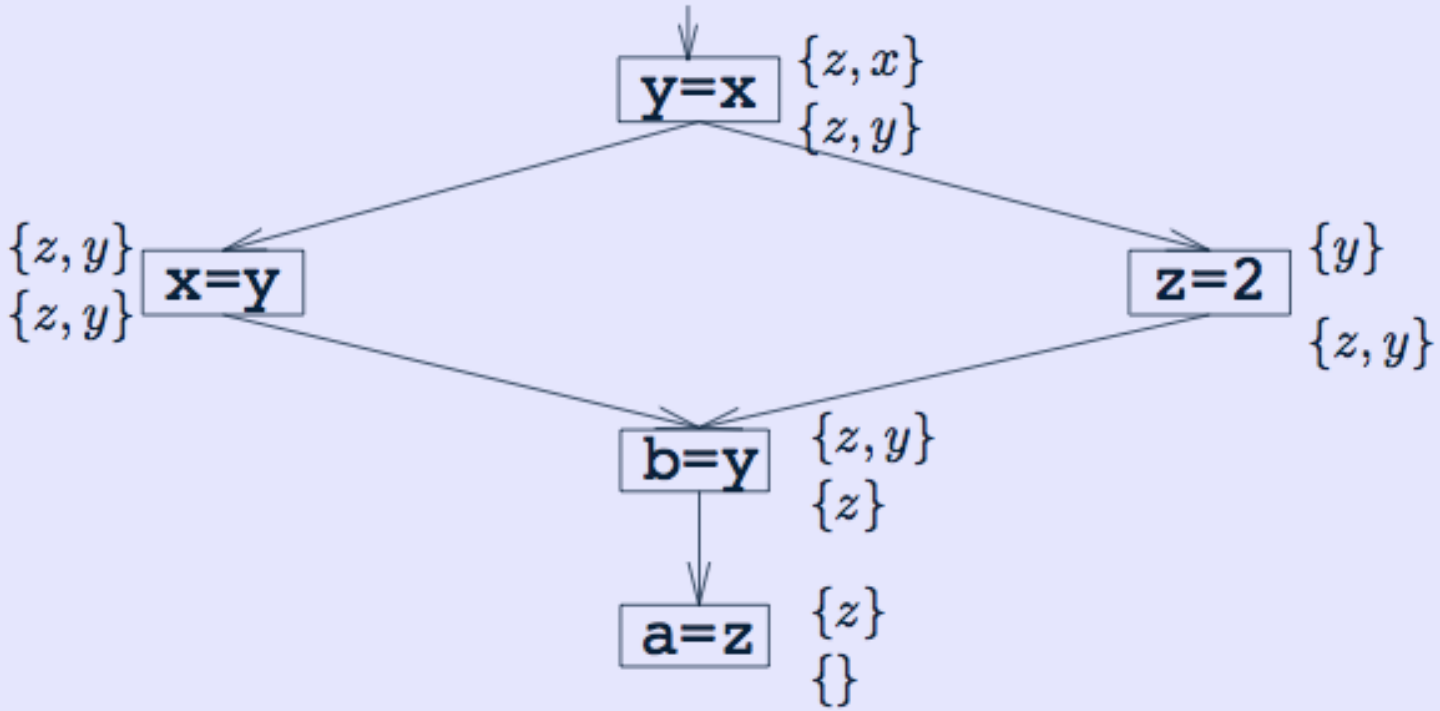
- Forward or backward?
- Decide what you are approximating.
  What is the domain's confluence operator?
  (Union or Intersection)
- Write equation for each kind of IR statement.
- State the starting approximation. (Initial value of each set)

HOWTO: Implementing Soot Flow Analysis

- Subclass ForwardFlowAnalysis or BackwardFlowAnalysis.
- Implement merge(), copy()
- Implement flow function: flowThrough()
- Implement initial values: newInitialFlow() and entryInitialFlow()
- Implement constructor
  (it must call doAnalysis())

Soot Example: Live Variable

A local variable $\mathbf{v}$ is live at s if there exists some statement s' using $\mathbf{v}$ and a control-flow path from s to s' free of definitions of $\mathbf{v}$.



Step1. Forward or Backward

Live variables is a backward flow analysis, since flow f^n computes IN sets from OUT sets.

In Soot, we subclass `BackwardFlowAnalysis`.

```
class LiveVariablesAnalysis
    extends BackwardFlowAnalysis
```

Step2. Abstraction Domain

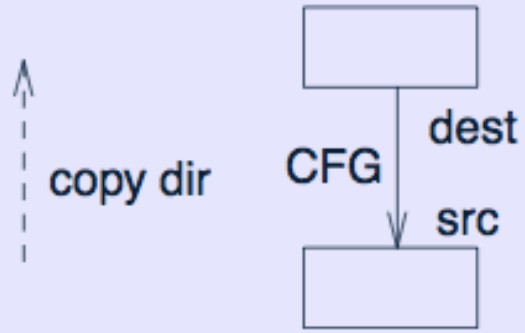
Domain for Live Variables: sets of `Locals`
e.g. {`x, y, z`}

- Partial order is subset inclusion
- Merge operator is union

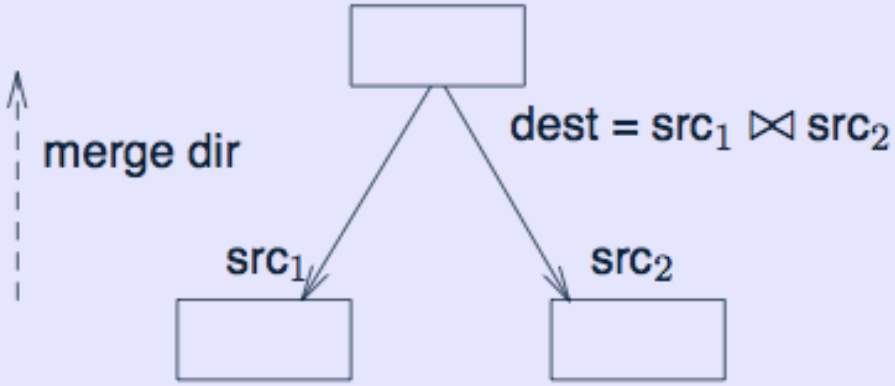
In Soot, we use the provided `ArraySparseSet` implementation of `FlowSet`.

Step 3. Implementing an Abstraction

Need to implement `copy()`, `merge()` methods:



`copy()` brings IN set to predecessor's OUT set.



`merge()` joins two IN sets to make an OUT set.

Step 3. Implementing an Abstraction

Signatures:

```
void merge(Object src1, Object src2,
           Object dest);
void copy(Object src, Object dest);
```

We delegate implementation to `FlowSet`.

Implementing Copy

```
protected void copy(Object src,
                    Object dest) {
HashSet<Local> sourceSet = (HashSet<Local>)src;
HashSet<Local> destSet = (HashSet<Local>)dest;
destSet.clear();
for(Local local : sourceSet)
    destSet.add(local);
}
```

Use `copy()` method from `FlowSet`.

Implementing Merge

In live variables, a variable `v` is live if there exists any path from `d` to `p`, so we use union.

Like `copy()`, use `FlowSet`'s `union`:

```
void merge(...) {
  // [cast Objects to FlowSets]
  src1Set.union(src2Set, destSet);
}
```

One might also use `intersection()`, or implement a more exotic merge.

Flow Equations

Goal: At a unit like `x = y * z`:

kill def `x`;

gen uses `y`, `z`.

How? Implement this method:

```
protected void flowThrough
                (Object srcValue,
                 Object u,
                 Object destValue)
```

Implementing Flow Function: Casting (1)

Soot's flow analysis framework is polymorphic. Need to cast to do useful work.

Start by:

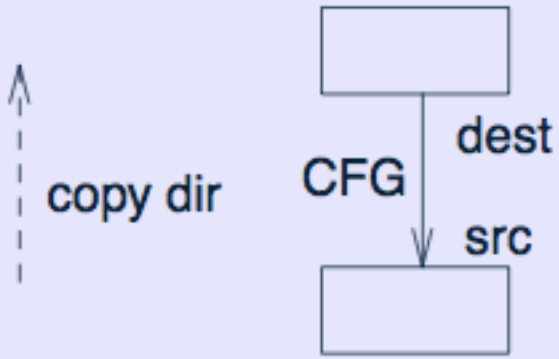
- casting `srcValue`, `destValue` to `FlowSet`.
- casting `u` to `Unit ut`.

In code:

```
FlowSet src = (FlowSet)srcValue,
        dest = (FlowSet)destValue;
Unit ut = (Unit)u;
```

Implementing Flow Function: Copying (2)

Need to copy `src` to `dest` to allow manipulation.




```
src.copy (dest);
```

Use `FlowSet` methods.

Implementing Flow Function: Removing Kills (3)

- $In(s) = Gen(s) \cup (Out(S) - Kill(s))$

```
// Take out kill set:
//  for each local v def'd in
//  this unit, remove v from dest
for (ValueBox box : ut.getDefBoxes())
{
  Value value = box.getValue();
  if( value instanceof Local )
    dest.remove( value );
}
```

Implementing Flow Function: Adding Gens (4)

- In(s) = Gen(s)

Goal: At a unit like `x = y * z`:

kill def `x`;

gen uses `y`, `z`.

```
// Add gen set
//  for each local v used in
//  this unit, add v to dest
for (ValueBox box : ut.getUseBoxes())
{
  Value value = box.getValue();
  if (value instanceof Local)
    dest.add(value);
}
```

Step 4. Initial Value.

- Soundly initialize IN, OUT sets prior to analysis.
  - Create initial sets

```
Object newInitialFlow()
    {return new ArraySparseSet();}
```

  - Create initial sets for exit nodes

```
Object entryInitialFlow()
    {return new ArraySparseSet();}
```

Want conservative initial value at exit nodes, optimistic value at all other nodes.

Step 5. Implement Constructor

```
LiveVariablesAnalysis(UnitGraph g)
{
    super(g);

    doAnalysis();
}
```

Causes the flow sets to be computed, using Soot's flow analysis engine.

In other analyses, we precompute values.

Enjoy: Flow Analysis Results

You can instantiate an analysis and collect results:

```
  LiveVariablesAnalysis lv =
      new LiveVariablesAnalysis(g);

// return SparseArraySets
// of live variables:
  lv.getFlowBefore(s);
  lv.getFlowAfter(s);
```

Roadmap

- Overview.
- Four Analysis Examples.
- Analysis Framework – Soot.
- **Theoretical Abstraction of Dataflow Analysis.**
- Inter-procedure Analysis.
- Taint Analysis.

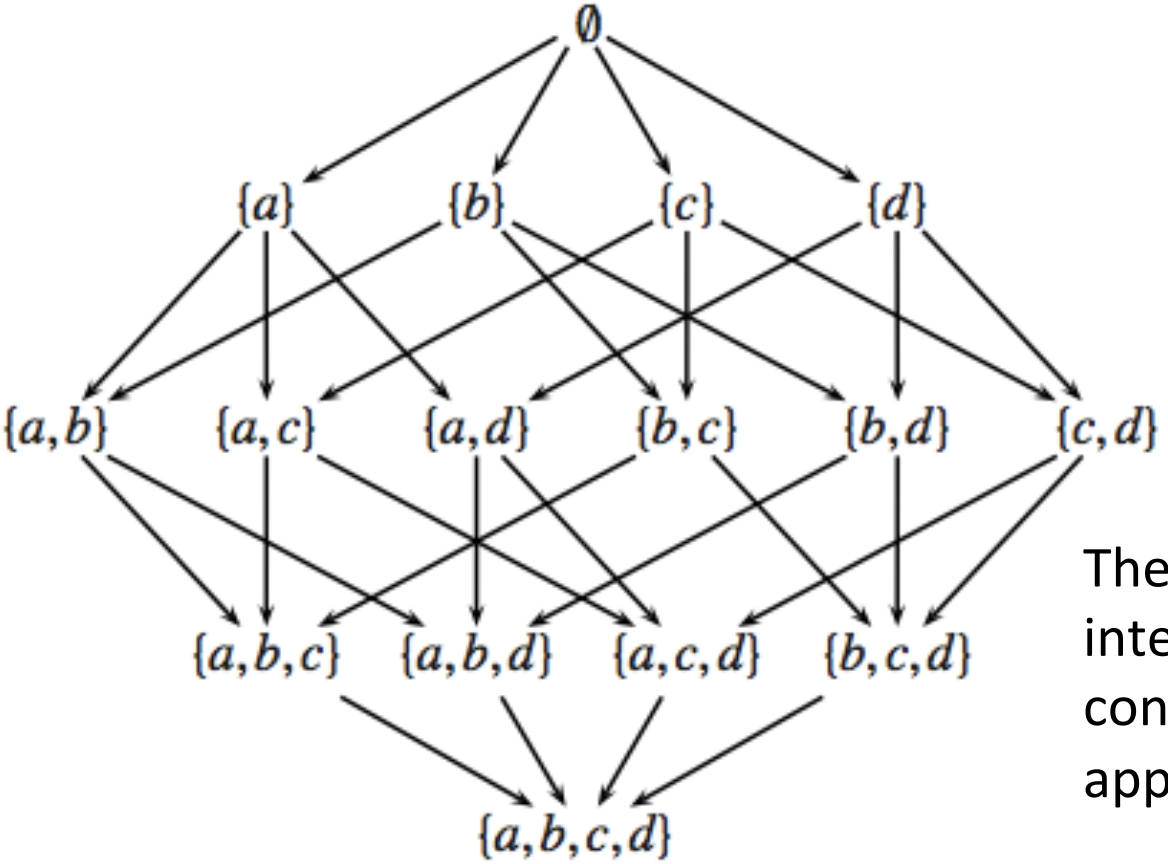
Theoretical Abstraction of Dataflow Analysis

- Model dataflow analysis is important:
  - Whether the algorithm will terminate.
  - Which analysis converges faster.
- There is an order between the values a data flow variable takes in two consecutive iterations.
- A general way to express an order between objects is to embed them in a mathematical structure called a lattice.

Lattice

- **A partial order** $\sqsubseteq$ on a set S is a relation over $S \times S$ that is
  - Reflexive. For all elements $x \in S : x \sqsubseteq x$.
  - Transitive. For all elements $x,y,z \in S : x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$.
  - Anti-symmetric. For all elements $x,y \in S : x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$.
- **A partially ordered set**, denoted by $(S , \sqsubseteq)$, is a set S with a partial order $\sqsubseteq$.
- **Lattice**: a partially ordered set with unique least upper bounds and greatest lower bounds.

Modeling Data Flow Values Using Lattices



The relation $\sqsubseteq$ can be interpreted as "a conservative (safe) approximation of".

The lattice for data flow values in live variables analysis

Modeling Flow Functions

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \tag{3.1}$$

$$Out_n = (In_n - Kill_n) \cup Gen_n \tag{3.2}$$

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} (In_p - Kill_p) \cup Gen_p & \text{otherwise} \end{cases} \tag{3.3}$$

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} f_p(In_p) & \text{otherwise} \end{cases} \tag{3.4}$$

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigsqcap_{p \in pred(n)} f_p(In_p) & \text{otherwise} \end{cases} \tag{3.5}$$

Two important properties of flow functions

DEFINITION 3.15 *A function* $f : L \mapsto L$ *is called monotonic iff*

$$\forall x, y \in L: \; x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

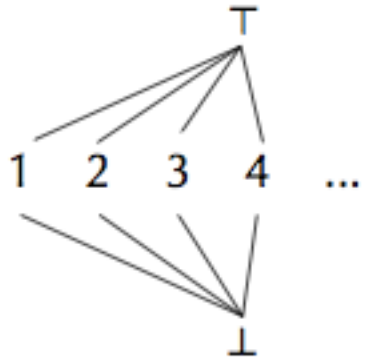
Determine if the analysis could terminate.

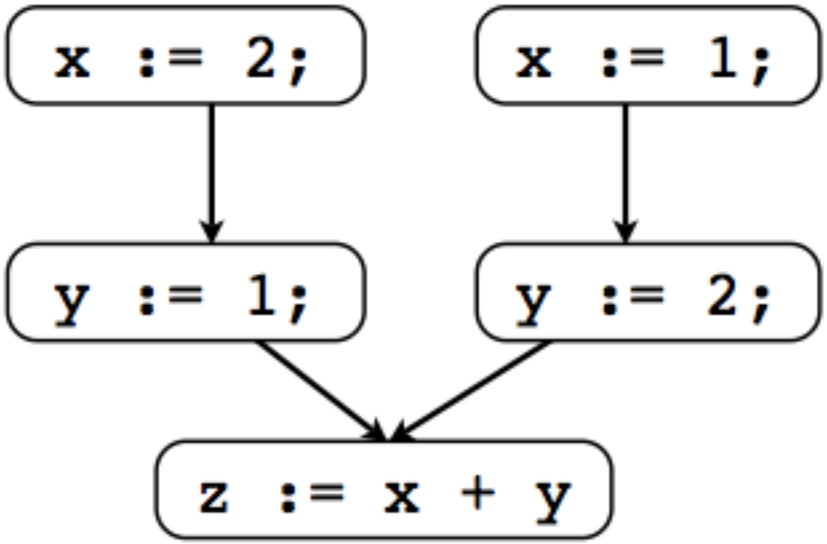
DEFINITION 3.16 *A function* $f : L \mapsto L$ *is called distributive iff*

$$\forall x, y \in L: \; f(x \sqcap y) = f(x) \sqcap f(y)$$

Determine if the analysis could run faster.

- Constant propagation





A: {x=2, y=1}

B: {x=1, y=2}

FlowFunction(A Π B) = {A,B,C unknown}

FlowFunction(A) Π FlowFunction (B) = {A,B Unknow, Z=3}

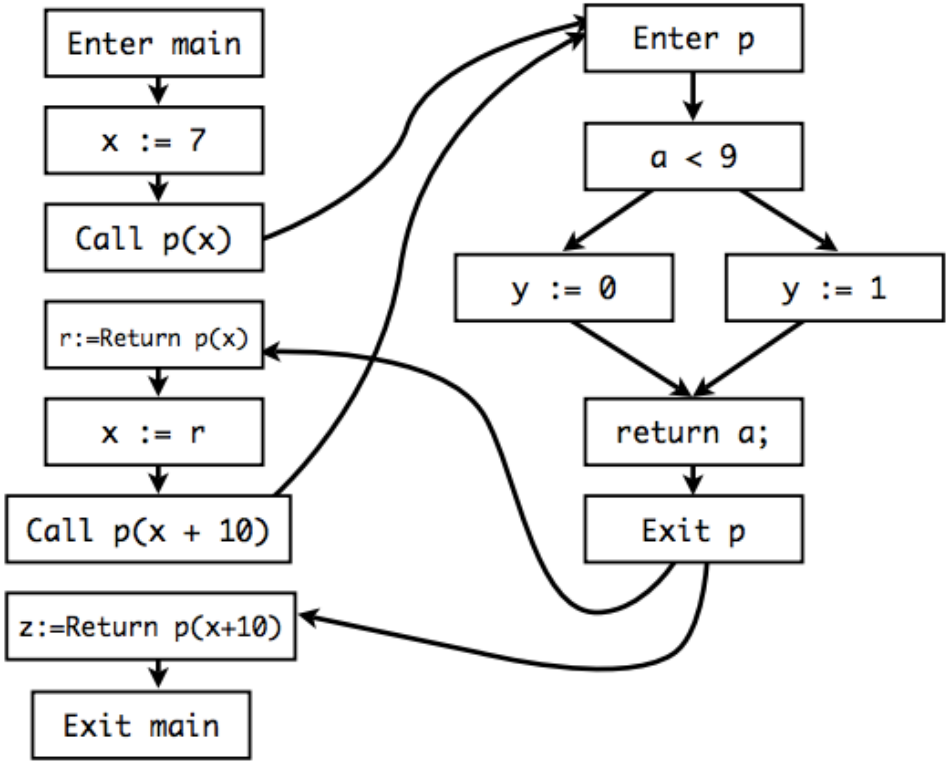
Roadmap

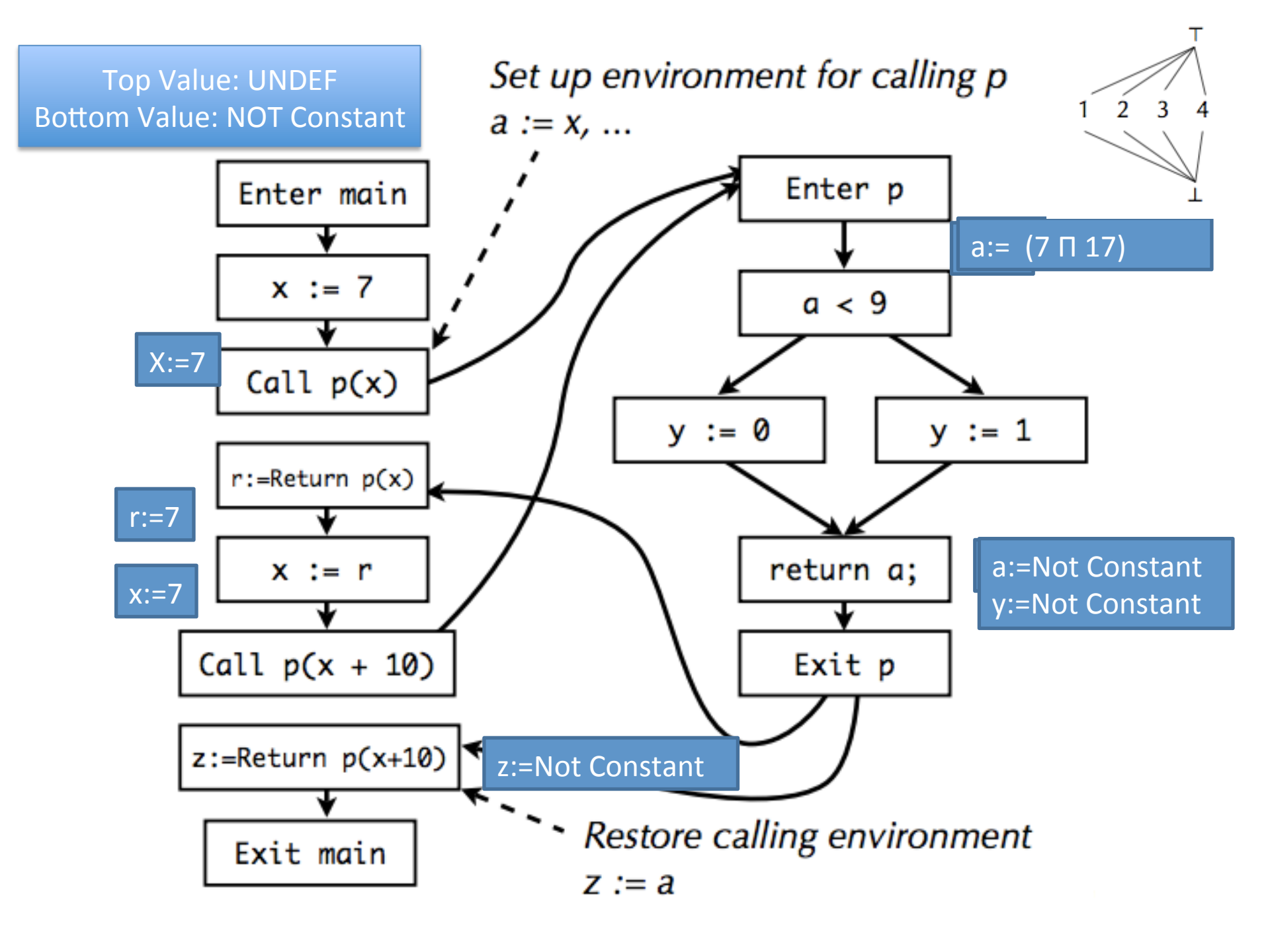
- Overview**.**
- Example.
- Theoretical Abstraction of Dataflow Analysis.
- **Inter-procedure Analysis.**
- Taint Analysis.
- Analysis Framework – Soot.

Inter-procedure Analysis

- How do we deal with procedure calls?
- Obvious idea: make one big CFG

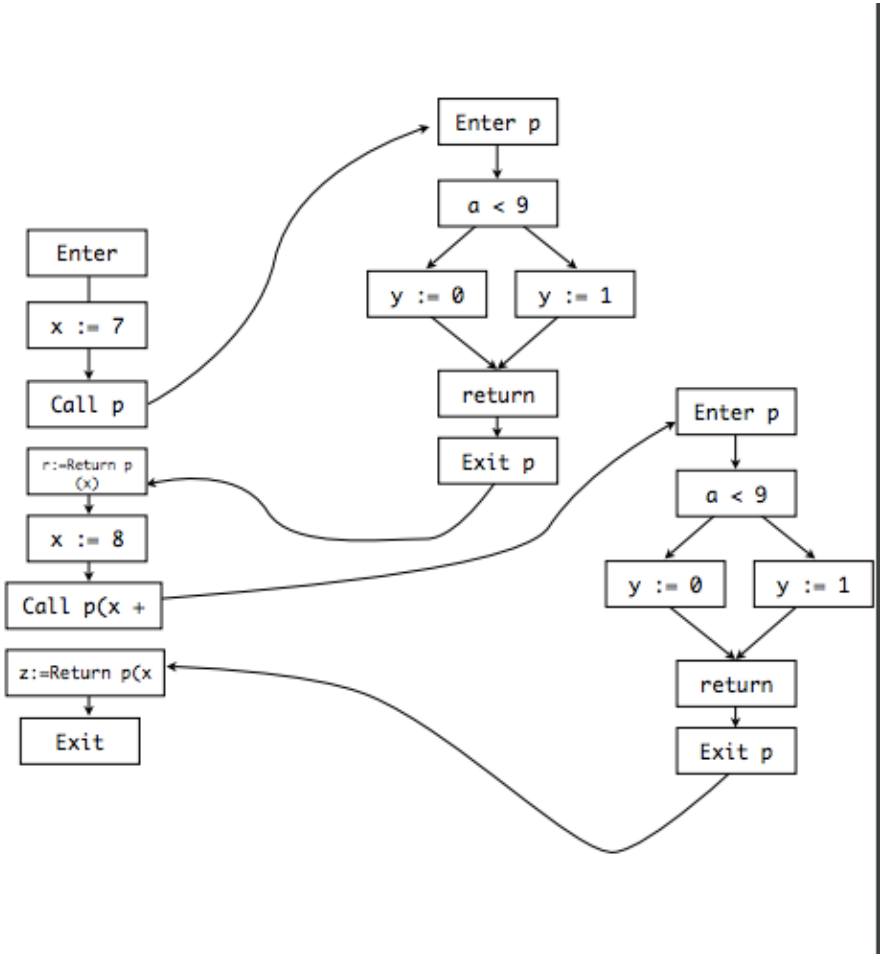
```
main() {
  x := 7;
  r := p(x);
  x := r;
  z := p(x + 10);
}

p(int a) {
  if (a < 9)
    y := 0;
  else
    y := 1;
  return a;
}
```




Top Value: UNDEF
Bottom Value: NOT Constant
Set up environment for calling p
a := x, ...
T
1 2 3 4
⊥
Enter main
x := 7
X:=7
Call p(x)
r:=Return p(x)
r:=7
x := r
x:=7
Call p(x + 10)
z:=Return p(x+10)
Exit main
Enter p
a:= (7 Π 17)
a < 9
y := 0
y := 1
return a;
a:=Not Constant
y:=Not Constant
Exit p
z:=Not Constant
Restore calling environment
z := a

A quick fix

- Inlining
  - Use a new copy of a procedure's CFG at each call site
- Problems? Concerns?
  - May be expensive! Exponential increase in size of CFG
    - p() { q(); q(); } q() { r(); r() } r() { … }
  - What about recursive procedures?
    - p(int n) { … p(n-1); … }
    - More generally, cycles in the call graph

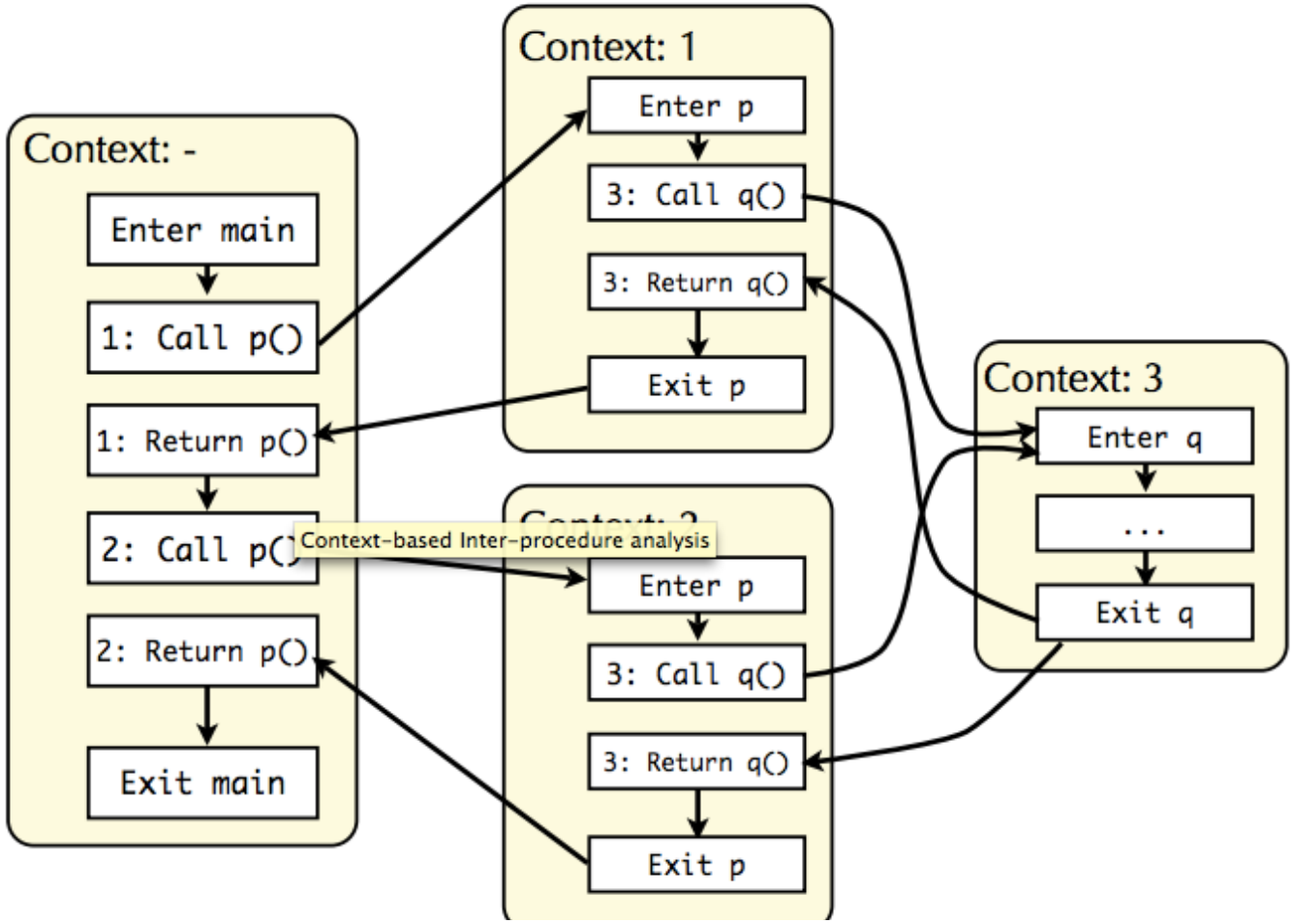


Context-based Inter-procedure analysis

- Solution: make a finite number of copies
- Use context information to determine when to share a copy
- Choice of what to use for context will produce different tradeoffs between precision and scalability
- Common choice:
  - Call site
  - Parameter value

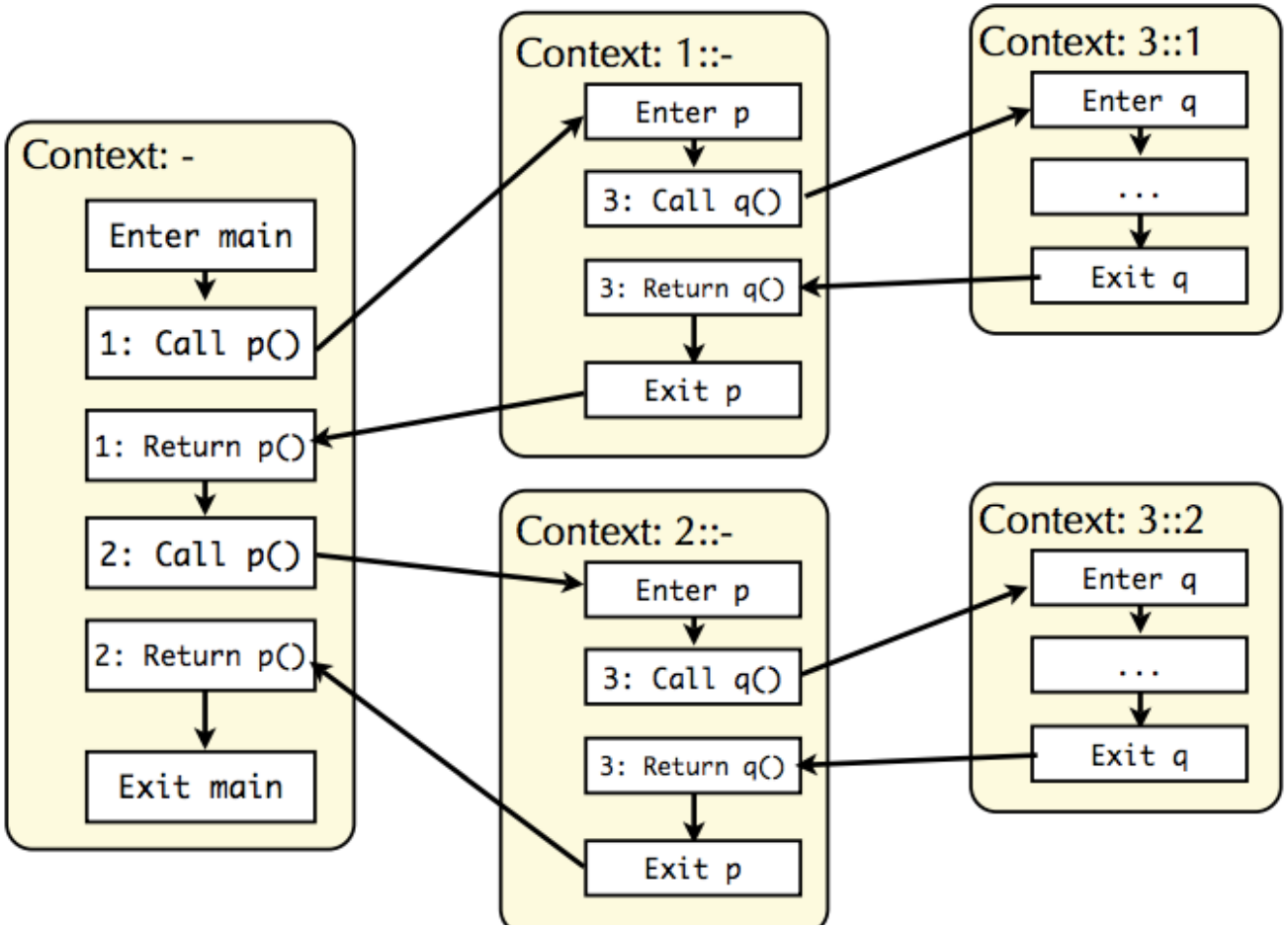
Based on Call Stack Depth 1

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```

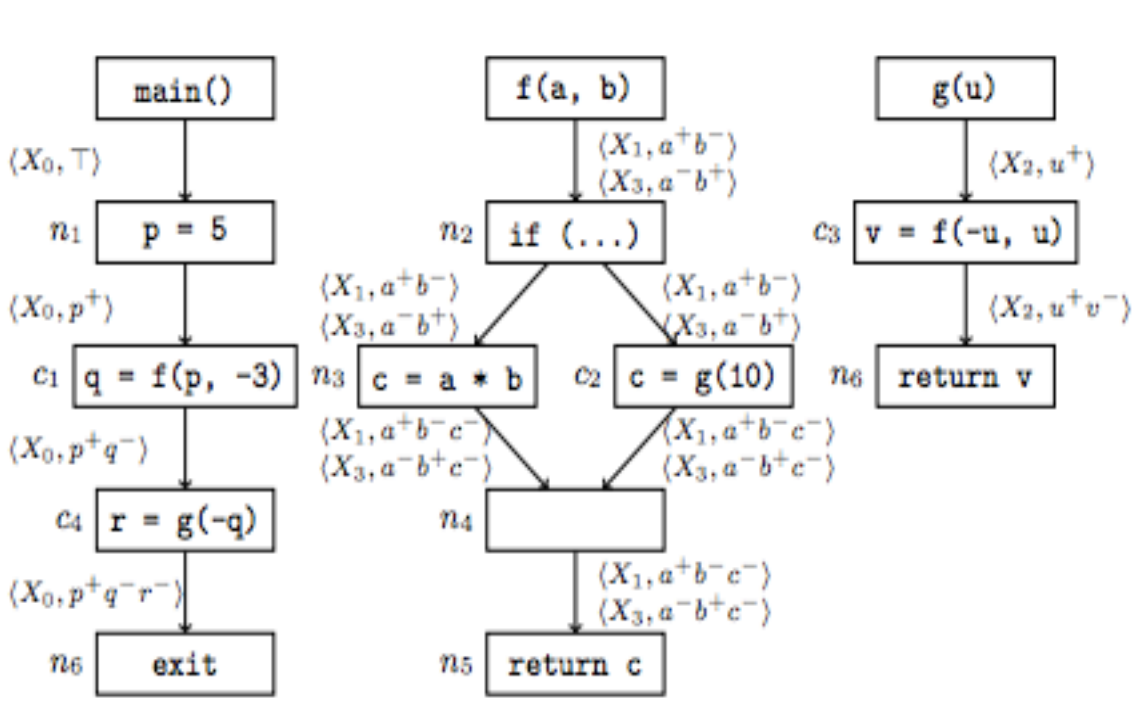


Based on Call Stack Depth 2

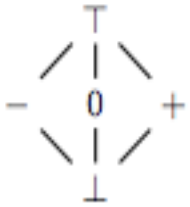
```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```



Based on Parameter Value



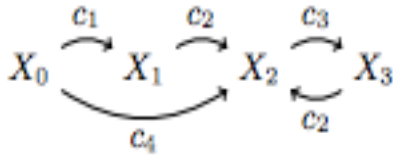
(a) Control flow graphs annotated with context-sensitive data flow values



(b) Lattice for a single variable

Context	Proc.	Entry	Exit
X_0	main	$\top$	$p^+q^-r^-$
X_1	f	a^+b^-	$a^+b^-c^-$
X_2	g	u^+	u^+v^-
X_3	f	a^-b^+	$a^-b^+c^-$

(c) Value contexts for the program



(d) Context transition diagram

Figure 2. A motivating example of a non-distributive sign-analysis performed on a program with mutually recursive procedures.

Roadmap

- Overview**.**
- Example.
- Theoretical Abstraction of Dataflow Analysis.
- Inter-procedure Analysis.
- **Taint Analysis**.
- Analysis Framework – Soot.

Taint Analysis

- Follow any application inside a debugger and you will see that data information is being copied and modified all the time. In another words, information is always moving.
- Taint analysis can be seen as a form of Information Flow Analysis.
  - Insert some kind of tag or label for data we are interested in. (taint the data)
  - Track the influence of the tainted object along the execution of the program.
  - Taint relevant data.
  - Obverse if it flows to sensitive functions (sink).

Taint Analysis

- Two usage in security:
  - **Finding information leakage.**
  - Finding program vulnerability.
- For information leakage:
  - If a data (variable) contains user secrets (e.g., location, contacts), we will taint such data.
  - Taint the variables whose data depend on tainted value. (e.g., a := b + x)
  - Observe if the tainted data will flow to functions that might send data to other places.

```
1  public class LeakageApp extends Activity{
2      private User user = null;
3      protected void onRestart(){
4          EditText usernameText = (EditText)findViewById(R.id.username);
5          EditText passwordText = (EditText)findViewById(R.id.password);
6          String uname = usernameText.toString();
7          String pwd = passwordText.toString();
8          this.user = new User(uname, pwd);
9      }
10     //Callback method; name defined in Layout-XML
11     public void sendMessage(View view){
12         if(user != null){
13             Password pwdObject = user.getPwdObject();
14             String password = pwdObject.getPassword();
15             String obfPwd = ""; //must track primitives
16             for(char c : password.toCharArray())
17                 obfPwd += c + "_"; //must handle concat.
18
19             String message = "User: " +
20                user.getUsername() + " | Pwd: " + obfPwd;
21             SmsManager sms = SmsManager.getDefault();
22             sms.sendTextMessage("+44 020 7321 0905", null,
23                 message, null, null);
24         }
25     }
26 }
```

Listing 1.1: Motivating example

Taint Analysis

- Two usage in security:
  - Finding information leakage.
  - **Finding program vulnerability (code injection).**
- Application vulnerability:
  - A lot of vulnerabilities are caused by unchecked input from user (attack) sent to sensitive functions.

“<script> alert(1)</script>

```
1: function postcomment($id, $title) {
2:     ...
3:     $title = urldecode($title);
4:     ...
5:     echo $title;
6:     ...
7: }
```

tainted

sensitive sink

```
1: if (...) {
2:     $entry = $_GET['entry'];
3:     ...
4:     $temp_file_name = $entry;
5:     ...
6: } else {
7:     ...
8:     $temp_file_name =
           stripslashes($_POST['file_name']);
9:     ...
10: }
11: ...
12: echo($temp_file_name);
```

XSS vulnerability

```
<?

function connect_to_db(){...}
function display_form(){...}
function grant_access(){...}
function deny_access(){...}

connect_to_db();

if
}
el

    // Run Query
    $query = "SELECT * FROM `login` WHERE `user`='$user' AND `pass`='$pass'";
    echo $query . "<br><br>";
    $SQL = mysql_query($query);

    // If user / pass combo found, grant access
    if(mysql_num_rows($SQL) > 0)
    grant_access();

    // Otherwise deny access
    else
    deny_access();
}

?>
```

SELECT * FROM `login` WHERE `user`= '' OR 'a' = 'a' AND `pass`= '' OR 'a' = 'a'

Buffer Overflow Vulnerability

```
#include <stdio.h>
int main(int argc, char **argv)
{
char buf[8]; // buffer for eight characters
gets(buf); // read from stdio (sensitive function!)
printf("%s\n", buf); // print out data stored in buf
return 0; // 0 as return value
}
```

Taint Analysis

- Two usage in security:
  - Finding information leakage.
  - **Finding program vulnerability (code injection).**
- Application vulnerability:
  - A lot of vulnerabilities are caused by unchecked input from user (attack) sent to sensitive functions.
  - If the source of a object X's value is untrusted, we say X is tainted.
  - Taint the variables whose data depend on tainted value. (e.g., a := b + x)
  - Observe if the tainted data will flow to dangerous functions that might lead to execution its parameters.

Taint Analysis Works

- Android App Information Leakage:
  - FlowDroid.
- JavaScript: Firefox Extension Vulnerability:
  - Bandhakavi, Sruthi, et al. "VEX: Vetting browser extensions for security vulnerabilities." Usenix Security. 2010.
- Php: Web Application Vulnerability:
  - Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities." Oakland, 2006

References

- Soot Tutorial: https://github.com/Sable/soot/wiki/Tutorials
- Interprocedural Data Flow Analysis in Soot using Value Contexts: https://arxiv.org/pdf/1304.6274.pdf
- Harvard Advanced Programming Language: http://www.seas.harvard.edu/courses/cs252/2011sp/
- Textbool: Data Flow Analysis: Theory and Practice: https://www.amazon.com/Data-Flow-Analysis-Theory-Practice/dp/0849328802
- Course: Professor Finddler's programming Languages seminar.
- Course: Professor Campanoni's code analysis and transformation.