

# Advanced Program Analyses for Object-oriented Systems

Dr. Barbara G. Ryder  
Rutgers University

<http://www.cs.rutgers.edu/~ryder>

<http://prolang.rutgers.edu/>

July 2007



## PROLANGS

- **Languages/Compilers and Software Engineering**
  - Algorithm design and prototyping
- **Mature research projects**
  - Incremental dataflow analysis, TOPLAS 1988, POPL'88, POPL'90, TSE 1990, ICSE97, ICSE99
  - Pointer analysis of C programs, POPL'91, PLDI'92
  - Side effect analysis of C systems, PLDI'93, TOPLAS 2001
  - *Points-to* and *def-use* analysis of statically-typed object-oriented programs (C++/Java) - POPL'99, OOPSLA'01, ISSTA'02, Invited paper at CC'03, TOSEM 2005, ICSM 2005
  - Profiling by sampling in Java feedback directed optimization, LCPC'00, PLDI'01, OOPSLA'02
  - Analysis of program fragments (i.e., modules, libraries), FSE'99, CC'01, ICSE'03, IEEE-TSE 2004



# PROLANGS

- **Ongoing research projects**
  - **Change impact analysis for object-oriented systems** PASTE'01, DCS-TR-533(9/03), OOPSLA'04, ICSM'05, FSE'06, IEEE-TSE 2007, ISSTA'07
  - **Robustness testing of Java web server applications**, DSN'01, ISSTA'04, IEEE-TSE 2005, Eclipse Wkshp OOPSLA'05, ICSE'07
  - **Analyses to aid performance understanding of programs built with frameworks** (e.g., Tomcat, Websphere) ISSTA'07
  - **Using more precise static analyses information** SE applications, SCAM'06, JSME'07, PASTE'07

# Course Outline

- **Lecture 1**
  - **Theoretical foundations of dataflow analysis**
  - **Issues in analyzing OO programs**
    - Polymorphism
    - Dynamic class loading and reflection
    - Use of libraries and frameworks
  - **What is reference analysis?**

## Course Outline

### • Lecture 2

- Type-based call graph construction
  - CHA, RTA
- Dimensions of analysis precision
- Context-insensitive reference analysis
  - Flow sensitivity, context sensitivity, field sensitivity
  - XTA, FieldSens
  - Client analyses: Side effects, devirtualization, thread/method-local objects

## Course Outline

### • Lecture 3

- Context-sensitive reference analyses of OO programs
  - K-CFA versus object-sensitive analysis (ObjSens)
  - Client analyses: cast check removal, devirtualization
- Dynamic analysis of OO programs
  - Finding 'hot methods' for JIT compilation through sampling

## Course Outline

- **Lecture 4**
  - Experiences with dynamic sampling for FDO
  - Optimizations for OO programs
    - Method inlining w & w/o guards
      - Pre-existence
    - Control-flow path splitting
    - Method specialization
    - Object layout for better cache performance
    - JIKES RVM online FDO experiments

## Course Outline

- **Lecture 5**
  - Analysis uses in testing and program understanding
  - Uses of analysis in software tools
    - Testing exception handling code
    - Interclass dependence analysis for class testing
    - Blended analysis for performance diagnosis

## Lecture 1 - Outline

- Motivation
- Theoretical foundations of dataflow analysis
  - Lattices, monotone DF frameworks, fixed point theorem,
    - Convergence, complexity, precision, safety properties
- How analysis of OO programs is different from classical (Fortran) analysis
  - Polymorphism
  - Dynamic class loading and reflection
  - Use of libraries and frameworks
- Call graph construction - enabling technology

ACACES-1 July 2007 © B6 Ryder

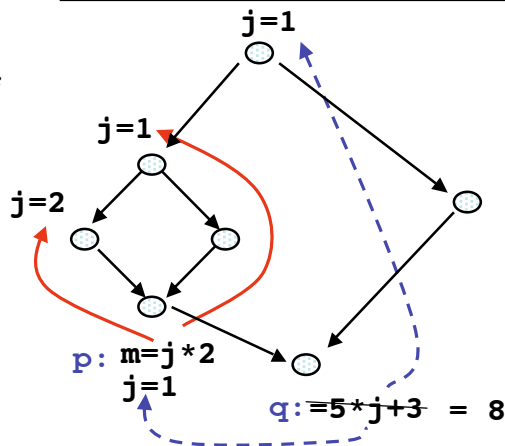


9

## Example

```
int f(){
  int j,k;
  j=1;
  if(...){j=1;
    if(...) j=2;
    k=j*2;
    p: m=j*2;
    j=1;}
  else {...}
  q: k=5*j+3;
  return k;
}
```

Are any of these expressions at  $p$  and  $q$ , compile-time constants?



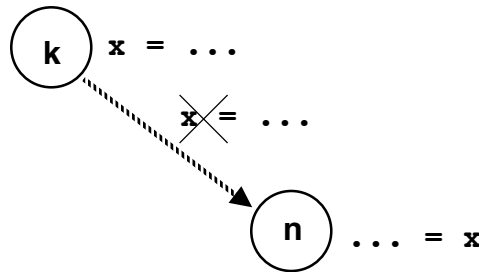
ACACES-1 July 2007 © B6 Ryder



10

## Reaching Definitions Dataflow Problem

- **Definition** A statement which may change the value of a variable
- A definition of a variable  $x$  at node  $k$  reaches node  $n$  if there is a definition-clear path from  $k$  to  $n$ .



ACACES-1 July 2007 © B6 Ryder



11

## Reaching Definitions Equations

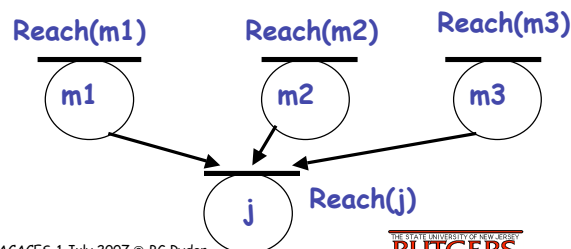
$$\text{Reach}(j) = \bigcup_{m \in \text{Pred}(j)} \{ \text{Reach}(m) \cap \text{pres}(m) \cup \text{dgen}(m) \}$$

where:

$\text{pres}(m)$  is the set of defs preserved through node  $m$

$\text{dgen}(m)$  is the set of defs generated at node  $m$

$\text{Pred}(j)$  is the set of immediate predecessors of node  $j$



ACACES-1 July 2007 © B6 Ryder



12

## Questions

- **How do we solve these dataflow eqns?**
  - How do we know that a solution exists?
  - How do we know how quickly a solution can be found?
- **How do we formulate other dataflow problems that are useful for code optimization?**
  - What do we need to define to formulate a dataflow analysis?
- **How do we define dataflow problems that involve method calls (interprocedural)?**

## Answers

- **Firm, mathematical foundations underlie dataflow analysis**
  - Lattice theory, partially ordered sets
  - Functions with specific properties to ensure convergence
    - Fixed point theorem provides solution procedure
- **Serves as underpinnings of all static analyses in compilation**
  - But not necessary to explain all analyses using this formalism

## Lattice Theory

- **Partial ordering  $\leq$** 
  - Relation between pairs of elements
  - Reflexive  $x \leq x$
  - Anti-symmetric  $x \leq y, y \leq x \Rightarrow x = y$
  - Transitive  $x \leq y, y \leq z \Rightarrow x \leq z$
- **Partially ordered set (Set  $S, \leq$ )**
- **0 Element  $0 \leq x, \forall x \in S$**
- **1 Element  $1 \geq \forall x \in S$**

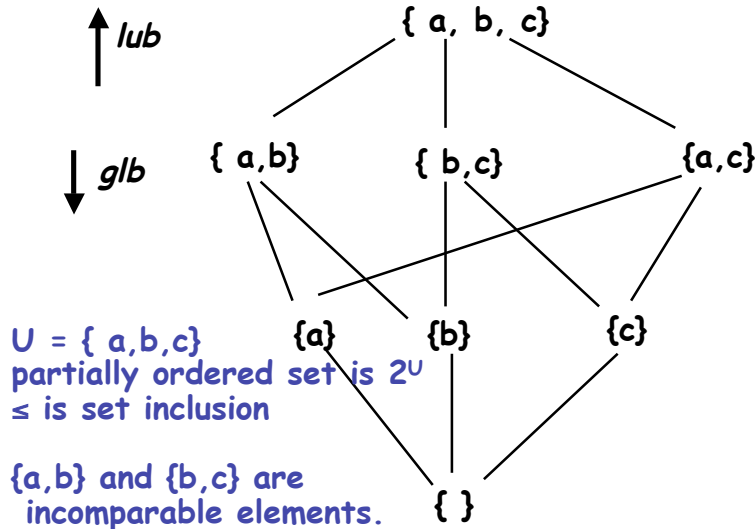
A partially ordered set need not have 0 or 1 element.

## Lattice Theory

- **Greatest lower bound (glb)**
  - $a, b \in$  partially ordered set  $S, c \in S$  is  $\text{glb}(a, b)$
  - if  $c \leq a$  and  $c \leq b$  then
  - for any  $z \in S, z \leq a, z \leq b \Rightarrow z \leq c$
- if glb is unique it is called the meet ( $\wedge$ ) of  $a$  and  $b$**
- **Least upper bound (lub)**
  - $a, b \in$  partially ordered set  $S, c \in S$  is  $\text{lub}(a, b)$
  - if  $c \geq a$  and  $c \geq b$  then
  - for any  $d \in S, d \geq a, d \geq b \Rightarrow c \leq d$ .
- if lub is unique is called the join ( $\vee$ ) of  $a$  and  $b$**



## Partially Ordered Set Example



ACACES-1 July 2007 © B6 Ryder



17

## Definition of a Lattice $(L, \wedge, \vee)$

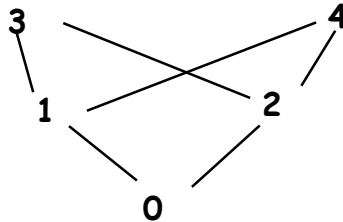
- $L$ , a partially ordered set under  $\leq$  such that every pair of elements has a unique glb (meet) and lub (join).
- A lattice need not contain an 0 or 1 element.
- A finite lattice must contain an 0 and 1 element.
- Not every partially ordered set is a lattice.
- If  $a \leq x, \forall x \in L$ , then  $a$  is 0 element of  $L$
- If  $x \leq a, \forall x \in L$ , then  $a$  is 1 element of  $L$

ACACES-1 July 2007 © B6 Ryder



18

## a partially ordered set, but not a lattice



There is no  $\text{lub}(3,4)$  in this  
partially ordered set  
so it is not a lattice.

## Examples of Lattices

- $H = (2^U, \cap, \cup)$  where  $U$  is a finite set
  - $\text{glb}(s_1, s_2)$  is  $(s_1 \wedge s_2)$  which is  $s_1 \cap s_2$
  - $\text{lub}(s_1, s_2)$  is  $(s_1 \vee s_2)$  which is  $s_1 \cup s_2$
- $J = (N_1, \text{gcd, lowest common multiple})$ 
  - partial order relation is integer divide on  $N_1$   
 $n_1 \mid n_2$  if division is even
  - $\text{lub}(n_1, n_2)$  is  $n_1 \vee n_2 = \text{lowest common multiple}(n_1, n_2)$
  - $\text{glb}(n_1, n_2)$  is  $n_1 \wedge n_2 = \text{greatest common divisor}(n_1, n_2)$

## Chain

- A partially ordered set  $C$  where, for every pair of elements

$c1, c2 \in C$ , either  $c1 \leq c2$  or  $c2 \leq c1$ .

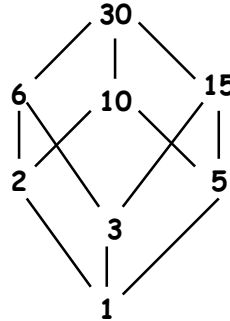
e.g.,  $\{ \} \leq \{a\} \leq \{a,b\} \leq \{a,b,c\}$

and from the lattice as shown here,

$$1 \leq 2 \leq 6 \leq 30$$

$$1 \leq 3 \leq 15 \leq 30$$

Lattices are used in dataflow analysis to argue the existence of a solution obtainable through fixed-point iteration.



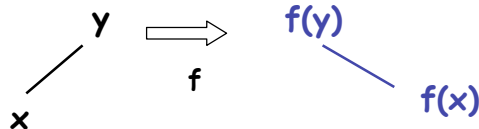
**Finite length lattice:** if every chain in lattice is finite

## Functions on a Lattice

- $(S, \leq)$  partially ordered set,  $f: S \rightarrow S$  is **monotonic** iff

$$\forall x, y \in S, x \leq y \Rightarrow f(x) \leq f(y)$$

- Monotonic** functions preserve domain ordering in their range values



- Distributive** functions allow function application to distribute over the meet

$$\forall x, y \in S, f(x) \wedge f(y) = f(x \wedge y)$$

## Fixed point theorem - Why it works?

### Intuition:

Given a 0 in lattice and **monotonic function**  $f$ ,  $0 \leq f(0)$ .

Apply  $f$  again and obtain

$$0 \leq f(0) \leq f(f(0)) = f^2(0)$$

Continuing,

$0 \leq f(0) \leq f^2(0) \leq f^3(0) \leq \dots \leq f^k(0) \leq f^{k+1}(0)$  for a finite chain lattice.

This is tantamount to saying

$\lim_{k \Rightarrow \infty} f^k(0)$  exists and is called the *least fixed point* of  $f$ ,

since  $f(f^k(0)) = f^{k+1}(0)$   
 $k \Rightarrow \infty$

## Fixed Point Theorem

Thm:  $f: S \rightarrow S$  monotonic function on poset  $(S, \leq)$  with a 0 element and finite length. The *least fixed point* of  $f$  is  $f^k(0)$  where

i.  $f^0(x) = x$ ,

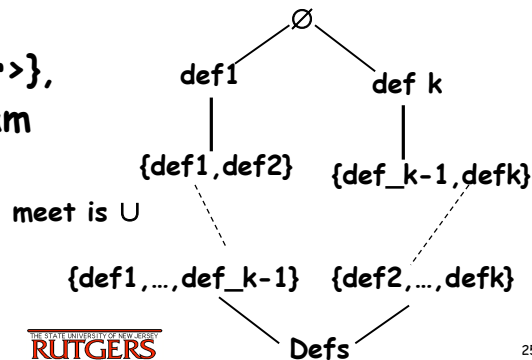
ii.  $f^{i+1}(x) = f(f^i(x))$ ,  $i \geq 0$ ,

iii.  $f^k(0) = f(f^k(0))$  and this is the smallest  $k$  for which this is true.

- For any  $p$  such that  $f(p)=p$ ,  $f^k(0) \leq p$ .
- Theorem justifies the iterative algorithm for global data flow analysis for lattices & functions with right properties
- Dual theorem exists for 1 element and *maximal fixed point* for  $k$  such that  $f^k(1) = f^{k+1}(1)$ .

## Reaching Definitions

- REACH meet operation is set union with partial order is  $\supseteq$  superset inclusion
  - Why? recall that the 0 element  $a$  is such that  $a \leq x = a, \forall x$  which means  $a$  is a superset of  $x$ !
- Defs =  $\{\langle \text{node}, \text{var} \rangle\}$ , all defs in program
- 0 element Defs
- 1 element is  $\emptyset$



ACACES-1 July 2007 © B6 Ryder



25

## How to solve? Worklist Algm

$$\text{Reach}(j) = \bigcup_{m \in \text{Pred}(j)} \{ \text{Reach}(m) \cap \text{pres}(m) \cup \text{dgen}(m) \}$$

Initialize all CFG nodes to  $\emptyset$ .

Put all nodes on the worklist  $W$ .

Loop: Do until  $W$  is empty{

    remove a node from the worklist  $W$ ;  
 calculate righthandside of above eqn;  
 compare result with  $\text{Reach}(j)$

    if result is different, {update  $\text{Reach}(j)$  and  
 put descendent nodes of  $j$  on worklist  $W$ }

}

//when terminates have correct reaching definitions  
 solution at each node

ACACES-1 July 2007 © B6 Ryder



26

## Monotone Dataflow Frameworks

- Formalism for expressing and categorizing data flow problems (Kildall, POPL'73)  $\langle G, L, F, M \rangle$ 
  - $G$ , flowgraph  $\langle N, E, \rho \rangle$
  - $L$ , (semi-)lattice with meet  $\wedge$ 
    - usually assume  $L$  has a 0 and 1 element
    - finite chains
  - $F$ , function space,  $\forall f \in F, f: L \rightarrow L$ 
    - Contains identity function
    - Closed under composition  $\forall f, g \in F, f \circ g \in F$
    - Closed under pointwise meet, if  $h(x) = f(x) \wedge g(x)$  then  $h \in F$
  - $M: E \rightarrow F$ , maps an edge to a corresponding transfer function that describes data flow effect of traversing that edge

## Function Properties that Guarantee a Solution

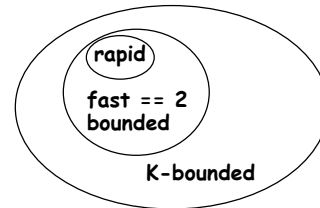
- **Monotonicity**
  - Defined as  $x \leq y \Rightarrow f(x) \leq f(y)$ .
  - Equivalent formulation of definition  
 $f(x \wedge y) \leq f(x) \wedge f(y)$
- **Distributivity**
  - If  $f(x \wedge y) = f(x) \wedge f(y)$  then  $f$  *distributive*
  - **Distributivity implies monotonicity**
  - Four classical bitvector problems are distributive

## Function Properties - Convergence

**K-bounded:** all contributions to MFP solution occur prior to Kth iteration

**Fast:** 1 pass around a cycle is enough to summarize its contribution to the dataflow solution (e.g., reflexive transitive closure is fast but not rapid)

**Rapid:** contribution of a cycle is independent of value at entry node; 1 pass around the cycle is enough. All classical bitvector problems are rapid

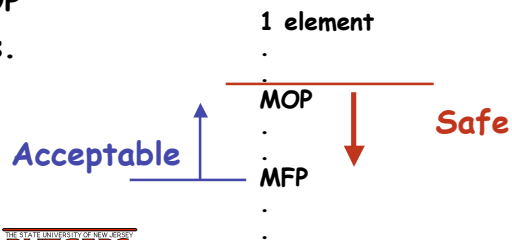


## MOP vs MFP

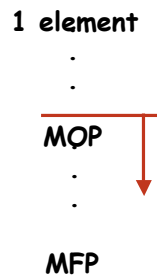
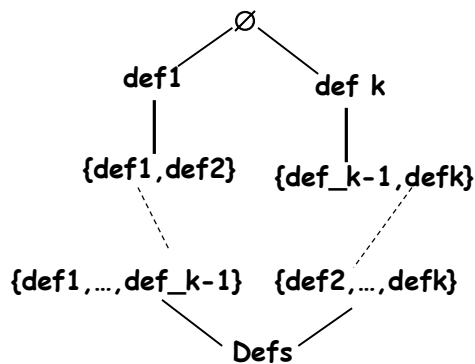
- If *distributive* functions define the dataflow problem, to obtain dataflow solution at node  $n$ , can gather information on paths (e.g.,  $P_1$ ,  $P_2$ ) simultaneously without loss of precision.
  - e.g.,  $f_{P_1}(0)$ ,  $f_{P_2}(0)$  needn't be calculated explicitly
- However, Kam and Ullman showed that this is not true for all *monotone* functions; Kam, Ullman, 1976, 1977
- Therefore, MFP only approximates MOP for general monotone functions that are not distributive.

## Safety of Dataflow Solution

- **Safe solution** underestimates the actual dataflow solution;  $x \leq MOP$  is an approximate solution
- **Acceptable solution** is one that contains a fixed point of the function,  $y \geq z$  where  $z$  is any fixed point.
- If they exist, **MOP is largest safe solution** and **MFP is smallest acceptable solution**.
- Between MFP and MOP are **interesting solutions**.



## Reaching Definitions



**Safe solutions contain the MOP**



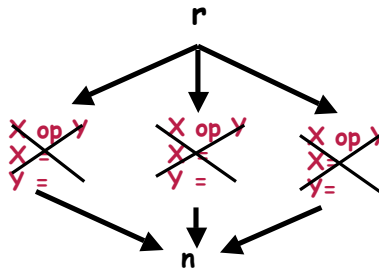
## Safe Solutions

- REACH: it is safe to err by saying a definition reaches when it DOES NOT REACH.
  - This may inhibit dead code elimination transformations
  - Since REACH functions are distributive, MOP=MFP here

## Available Expressions

- An expression  $X \text{ op } Y$  is *available* at program point  $n$  if EVERY path from program entry to  $n$  evaluates  $X \text{ op } Y$ , and after every evaluation prior to reaching  $n$ , there are NO subsequent assignments to  $X$  or  $Y$ .

Used to enable common subexpression elimination



## Available Expressions Equations

$$\text{Avail}(j) = \bigcap_{m \in \text{Pred}(j)} \{ \text{Avail}(m) \cap \text{epres}(m) \cup \text{egen}(m) \}$$

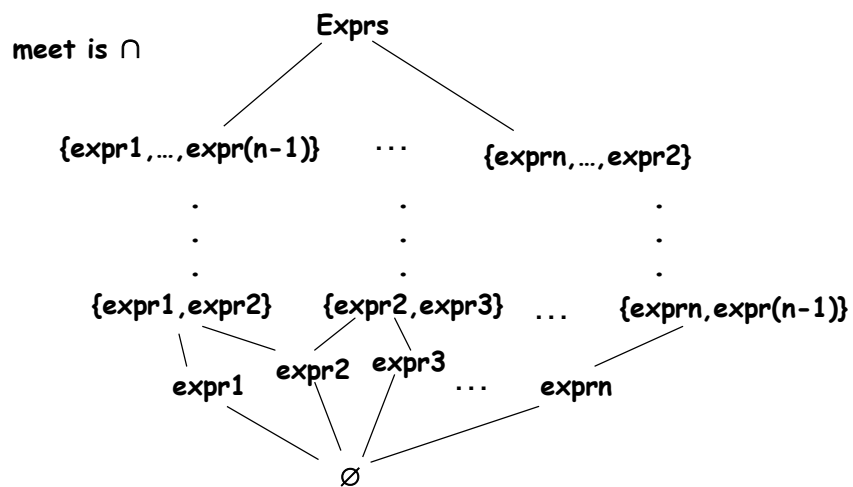
where:

**epres(m)** is the set of expressions preserved through node *m*

**egen(m)** is the set of (downwards exposed) expressions generated at node *m*

**pred(j)** is the set of immediate predecessors of node *j*

## Available Expressions



## Safe Solutions

- **AVAIL**: it is safe to err by saying an expression is **NOT AVAILABLE** when it might be.
  - This may inhibit *common subexpression elimination* transformations
  - Since **AVAIL** functions are distributive, **MOP=MFP** here

## How is analysis of OOPLs different?

- Domain: Java, C++, C# like languages

**Fortran:**  
Fixed call structure

Limited polymorphism  
of primitive types

Whole program analysis  
easy because all libraries  
necessary for compilation

**OOPLs:**  
Dynamic dispatch

Polymorphism (i.e., subtyping)

Use of libraries and frameworks

Dynamic statements affecting  
execution  
(e.g., reflective calls, dynamic  
class loading)

## Reference Analysis

- *Determines information about the set of objects to which a reference variable or field may point during program execution*
- **An enabling analysis for OOPLs**
  - Its precision affects precision of subsequent analysis clients (e.g., side effects)
  - Need to find the right cost/benefit tradeoff for particular problem

## Reference Analysis

- **OOPLs need type information about objects to which reference variables can point to resolve dynamic dispatch**
- **Often data accesses are indirect to object fields through a reference, so that the set of objects that might be accessed depends on which object that reference can refer at execution time**
- **Need to pose this as a compile-time program analysis with representations for reference variables/fields, objects and classes.**

## Reference Analysis enables...

- **Construction of possible calling structure of program - call graph**
  - Dynamic dispatch of methods based on runtime type of receiver `x.f();`
- **Understanding of possible dataflow in program**
  - Indirect side effects through reference variables and fields `r.g=`
- **Uses of call graph**
  - e.g., Program slicing, obtaining method coverage metrics for testing, heap optimization, etc

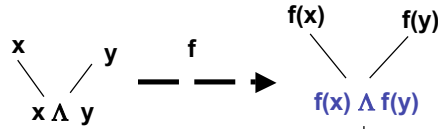
## Uses of Reference Analysis Information in Software Tools

- **Program understanding tools**
  - Semantic browsers
  - Program slicers
- **Software maintenance tools**
  - Change impact analysis tools
- **Testing tools**
  - Coverage metrics

## Analyses to be Discussed

- **Lecture 2:**
  - Type hierarchy-based reference analyses
    - CHA, RTA
  - Incorporating flow
    - FieldSens (Andersen-based points-to)
- **Lecture 3:**
  - Incorporating flow and calling context
    - 1-CFA
    - ObjSens (object-sensitive)

## Monotonicity



$f: L \rightarrow L$   
 $x \Delta y \leq x$   
 $x \Delta y \leq y$   
 by defn of meet of  
 $x, y$ ; and

$f(x) \Delta f(y) \leq f(x)$   
 $f(x) \Delta f(y) \leq f(y)$   
 by defn of meet of  
 $f(x), f(y)$ .

$f(x \Delta y) \leq f(x)$   
 $f(x \Delta y) \leq f(y)$   
 by monotonicity of  $f$

$f(x \Delta y) \leq f(x) \Delta f(y)$   
 by defn meet of  $f(x), f(y)$

Therefore,  $x \leq y \Rightarrow f(x) \leq f(y)$  (1)  
 implies  
 $f(x \Delta y) \leq f(x) \Delta f(y)$  (2).

## Monotonicity, cont.

Show  $f(x \Delta y) \leq f(x) \Delta f(y)$  (2) implies  $x \leq y \Rightarrow f(x) \leq f(y)$  (1)  
 Then we know these two definitions of monotonicity are equivalent.

Assume  $x \leq y$ . Then  $x \Delta y = x$  by defn of meet.

$f(x \Delta y) = f(x) \leq f(x) \Delta f(y)$  which is given.

Then  $f(x) \Delta (f(x) \Delta f(y)) = f(x)$  by defn of meet.

But  $(f(x) \Delta f(x)) \Delta f(y) = f(x) \Delta f(y) = f(x)$  by associativity of meet

Therefore,  $f(x) \leq f(y)$  by defn of meet.

So (2) implies (1).

Therefore, these definitions of monotonicity are equivalent.

## Available Expressions

- lattice is  $2^{\text{Exprs}}$  where Exprs is set of all binary expressions in program
- Partial order is  $\subseteq$  (subset inclusion) so meet is  $\cap$
- $\langle \text{Exprs}, \text{Exprs}, \dots, \text{Exprs} \rangle$  is 1 element
- $\langle \emptyset, \emptyset, \dots, \emptyset \rangle$  is 0 element
- From the data flow equations for AVAIL, we know that if a set of dataflow facts  $X$  is true on entry to a flowgraph node  $n$ , then  $f(X)$  is true on each exit edge of  $n$  where

$$f(X) = \text{epres}(n) \cap X \cup \text{egen}(n)$$

$f$  is called the *transfer function* for AVAIL

## Available Expressions

- Cross product lattice is
  - $(2^{\text{Exprs}}, 2^{\text{Exprs}}, \dots, 2^{\text{Exprs}})$  with  $n$  components where  $n$  is number of nodes in the cfg and  $\leq'$  is  $\leq$  component-wise
- Since Avail equation at a node can be expressed thusly,
 
$$\text{Avail}(j) = \bigcap_{m \in \text{Pred}(j)} \{ \text{Avail}(m) \cap \text{epres}(m) \cup \text{egen}(m) \}$$
  - AVAIL ( $j$ ) is the solution at entry of node  $j$  and  $f(\text{AVAIL}(j))$  is solution at exit of node  $j$ ,
  - $g_j = \bigcap_{m \in \text{Pred}(j)} f(g_m)$



## Available Expressions

- Can you show  $g_i$  monotone?  
 $g_i : (2^{\text{Exprs}}, 2^{\text{Exprs}}, \dots, 2^{\text{Exprs}}) \rightarrow 2^{\text{Exprs}}$
- Then this induces the monotonicity of  $F$ ,  
 $F = (g_1, \dots, g_n)$
- Application of dual of fixed point theorem here to find the maximal fixed point. Iterate down from the 1 element.
  - Initialize  $\rho$  to  $\emptyset$ , all other cfg nodes to Exprs.