## Dataflow Analysis: Non-distributive Analyses, Approximations

## Announcements

- HW1 due
- HW2 is out
  - Get started with your Submitty git repos, Soot and Jimple, etc.
  - We'll cover class analysis (RTA, XTA, etc.) on Thursday



## Outline of Today's Class

- One more note on Worklist algorithm
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations



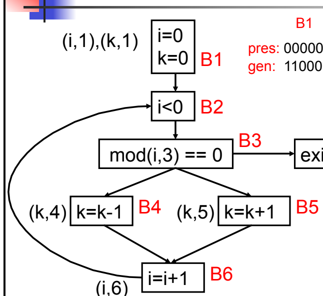
## Worklist Algorithm for Forward Dataflow Problems

/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = $f_1$(in(1))
for m := 2 to n do in(m) = **0**; out(m) = $f_m$(**0**)
W = {2,…,n} /* put every node but 1 on the worklist */

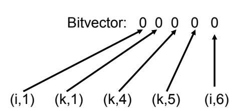
while W ≠ Ø do {
  remove j from W
  in(j) = **V** { out(i) | i is predecessor of j }
  out(j) = $f_j$(in(j))
  if out(j) changed then
    W = W U { k | k is successor of j }
}



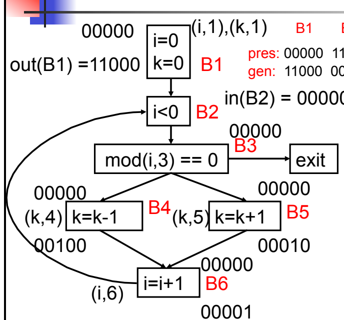
## Example. *Reach* with Bitvectors




| | B1 | B2 | B3 | B4 | B5 | B6 |
|---|---|---|---|---|---|---|
| pres: | 00000 | 11111 | 11111 | 10001 | 10001 | 01110 |
| gen: | 11000 | 00000 | 00000 | 00100 | 00010 | 00001 |



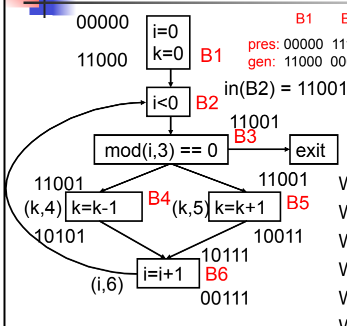

## Initialization




| | B1 | B2 | B3 | B4 | B5 | B6 |
|---|---|---|---|---|---|---|
| pres: | 00000 | 11111 | 11111 | 10001 | 10001 | 01110 |
| gen: | 11000 | 00000 | 00000 | 00100 | 00010 | 00001 |

## Iteration

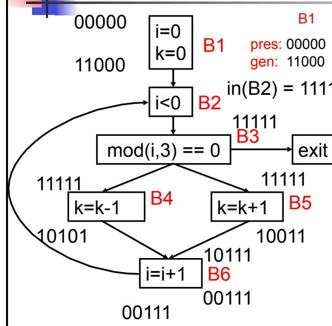
| | B1 | B2 | B3 | B4 | B5 | B6 |
|---|---|---|---|---|---|---|
| pres: | 00000 | 11111 | 11111 | 10001 | 10001 | 01110 |
| gen: | 11000 | 00000 | 00000 | 00100 | 00010 | 00001 |




W = { B2, B3, B4, B5, B6 }
W = { B3, B4, B5, B6 }
W = { B4, B5, B6 }
W = { B5, B6 }
W = { B6 }
W = { } W = { B2 }

## Iteration

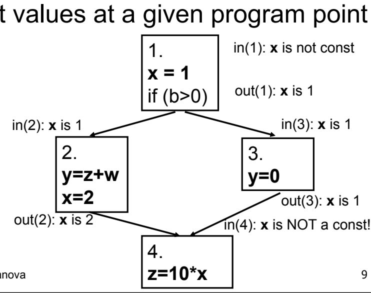
| | B1 | B2 | B3 | B4 | B5 | B6 |
|---|---|---|---|---|---|---|
| pres: | 00000 | 11111 | 11111 | 10001 | 10001 | 01110 |
| gen: | 11000 | 00000 | 00000 | 00100 | 00010 | 00001 |




W = { B2 }
W = { B3 }
W = { B4, B5 }
W = { B5, B6 }
W = { B6 }
W = { }

## Constant Propagation (Simple)

- Problem statement: What variables always hold constant values at a given program point
- Example:






## Aside: Defining an Analysis

- Define program syntax
  - In practice, we deal with a lot more than the simple abstraction ☺
- Define property space
  - The *abstract program state* that approximates the concrete program state
- Define transfer function space over syntax
  - Symbolically execute program over abstract state



## Aside: Defining an Analysis

- If property space has desired properties
  - is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - merge operator V is the join of **L**

  and
- Function space **F: L→ L** is monotone

then analysis fits the monotone dataflow framework and can be solved using the worklist algorithm



## Constant Propagation: Syntax

*S* ::= *S*; *S* | **while (b) {** *S* **}** | **if (b) {** *S* **} else {** *S* **}**
| **x =** *V* | **x =** *V Op V*
*Op* ::= + | - | * | /
*V* ::= **x** | **y** | **z** | **C**

- **x, y, z** are program variables
- **C** is constant
- We have to define transfer functions for **x =** *V* and **x =** *V Op V*

## Constant Propagation: Property Space

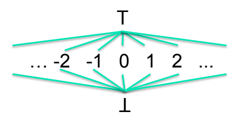
- Associate one of the following values with variable **x** at each program point

| value | meaning |
|---|---|
| **1** (or T) | **x** is NOT a constant |
| **C** | **x** has constant value **C** |
| **0** (or ⊥) | **x** is unknown |



## Constant Propagation: Lattice

- Lattice $\mathbf{L_x}$,≤




- Dataflow lattice **L** is the product lattice of $\mathbf{L_x}$
  - **l1,l2** in **L**, **l1** ≤ **l2** iff $\mathbf{l1_x} \le \mathbf{l2_x}$ for every variable **x**
  - **l1** V **l2** amounts to $\mathbf{l1_x}$ V $\mathbf{l2_x}$ for every variable **x**
  - Merge operator is join of **L**
- Does the product lattice satisfy the ACC?



## Product Lattice

- E.g.,

<**x=**⊥**, y=1, z=**T>, <**x=1, y=2, z=3**>, etc. are lattice elements

- E.g.,

<**x=1, y=2, z=**T> **≤** <**x=**T**, y=2, z=**T>

- E.g.,

<**x=1, y=3, z=**T> **V** <**x=**T**, y=2, z=**T> = <T, T, T>



## Constant Propagation: Transfer Functions

- **j: x = C**

  $f_j$: kill **x** → **val**, generate **x** → **C**

- **j: x = y**

  $f_j$: kill **x** → **val**, add **x** → **val'**, s.t. **y** → **val'** in in(j). **val** and **val'** are one of
  - ⊥: bottom (unknown)
  - C: constant
  - T: top (not a constant)



## Constant Propagation: Transfer Functions

- **j: x =** $V_1$ *Op* $V_2$

  $f_j$: kill: **x** → **val**

  gen:

If $V_1$ → $\mathbf{c_1}$ and $V_2$ → $\mathbf{c_2}$ in in(j), then **x** → $\mathbf{c_1}$ *Op* $\mathbf{c_2}$

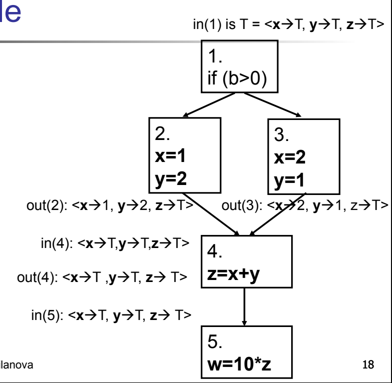
else if $V_1$ → T or $V_2$ → T in in(j), then **x** → T

else **x** → ⊥

- Next, we'll argue monotonicity which would give us that Constant Propagation is solvable by the Worklist algorithm



## Example

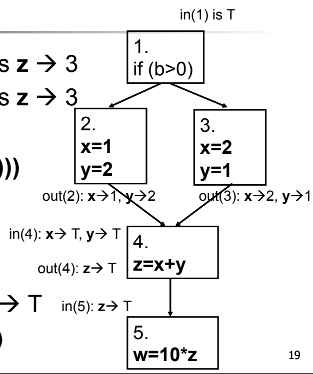


## Not Distributive! A Counter Example

- $f_4(f_2(f_1(T)))$ computes **z** → 3
- $f_4(f_3(f_1(T)))$ computes **z** → 3
- Thus, MOP at 5

$f_4(f_2(f_1(T))) \vee f_4(f_3(f_1(T)))$

computes **z** → 3



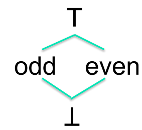

MFP at 5 computes **z** → T

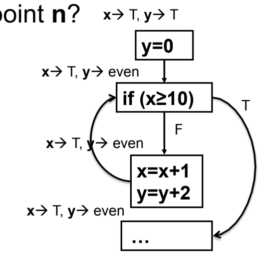
(i.e., **z** is NOT a const)



## More Product Lattices

- Problem statement: Is integer variable **x** odd or even at program point **n**?
- $L_x$:

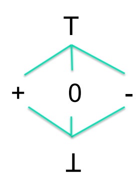





## More Product Lattices

- Problem statement: What sign does a variable hold at a given program point, i.e., is it positive, negative, or 0
- $L_x$:




E.g., **< x=+,y=**T**,z=0 >**



## Points-to Analysis

- Problem statement: What memory locations may a pointer variable point to?
- Many applications!
  - Enables compiler optimizations

| | |
|---|---|
| 1. a = 1; | 1. a = x*y*z+x; |
| 2. *p = b; | 2. *p = b; |
| 3. s = a*a; | 3. s = x*y*z+x; |

  - Static debugging tools, static taint analysis tools



## Points-to Analysis: Example

Example 1:

```
int a, b;
int *p1, *p2;
p1 = &a;
p2 = p1;
*p2 = 1;
```

Example 2:

```
int a, b = 15;
int *p1, *p2;
int **p3;
p3 = &p1;
p1 = &a;
p2 = *p3;
*p2 = b;
```



## Points-to Analysis: Syntax

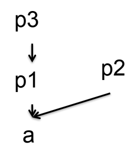
- Assume the following 4 simple statements

| | |
|---|---|
| (1) address taken | **p = &q** |
| (2) propagation | **p = q** |
| (3) indirect read | **p = *q** |
| (4) indirect write (update) | ***p = q** |

- One can transform any program into a sequence of statements of these kinds

## Points-to Analysis: Property Space

- Lattice **L,≤**
  - Lattice of the subsets over all edges **p → q** where **p** and **q** are program variables
  - … or in simpler terms, lattice elements are points-to graphs, e.g.,




  - **V** is points-to graph union
  - **0** of L is empty graph
  - **1** of L is complete graph



## Points-to Analysis: Transfer Functions

(1) $\mathbf{f}_{p=\&q}$: "kill" all points-to edges from **p**, and "generate" a new points-to edge from **p** to **q**

(2) $\mathbf{f}_{p=q}$: "kill" all points-to edges from **p**; "generate" new points-to edges from **p** to every **x**, such that **q** points to **x** in incoming points-to graph in(j)

(3) $\mathbf{f}_{p=*q}$: "kill" all points to edges from **p**; "generate" new points-to edges from **p** to every **x**, s.t. there is **y** where **q** points to **y**, and **y** points to **x** in in(j)

(4) $\mathbf{f}_{*p=q}$: Do not kill! Can you think of a reason why? "Generate" new points-to edges from every **y** to every **x**, such that **p** points to **y** and **q** points to **x**



## The Problem with Updates

- Updates (4) $\mathbf{f}_{*p=q}$ (also known as destructive updates) … are a pain
- If we drop (4) from our language we get

| | |
|---|---|
| (1) p = &a | **p = cons(a,null)** |
| (2) p = q | **p = q** (actual-to-parameter assignment) |
| (3) p = *q | **p = car(q)** |

- Research problems!



## Points-to Analysis is Monotone

- To argue monotonicity we must show that if $\mathbf{Pt_1}$ is **≤** (subset of) $\mathbf{Pt_2}$, then $\mathbf{f(Pt_1)} \leq \mathbf{f(Pt_2)}$ for each transfer function **f**

(1) $Pt_1 \leq Pt_2$ then $\mathbf{f}_{p=\&q}(Pt_1) \leq \mathbf{f}_{p=\&q}(Pt_2)$
(2) $Pt_1 \leq Pt_2$ then $\mathbf{f}_{p=q}(Pt_1) \leq \mathbf{f}_{p=q}(Pt_2)$
(3) $Pt_1 \leq Pt_2$ then $\mathbf{f}_{p=*q}(Pt_1) \leq \mathbf{f}_{p=*q}(Pt_2)$
(4) $Pt_1 \leq Pt_2$ then $\mathbf{f}_{*p=q}(Pt_1) \leq \mathbf{f}_{*p=q}(Pt_2)$

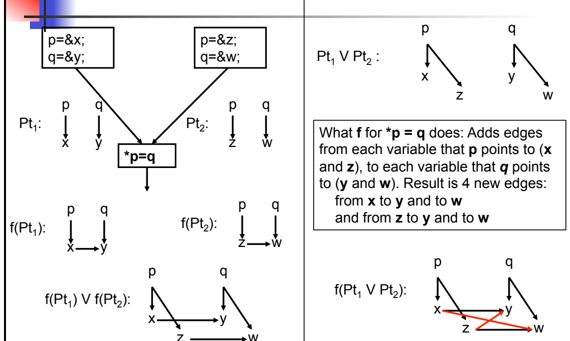


## … but it is not distributive!

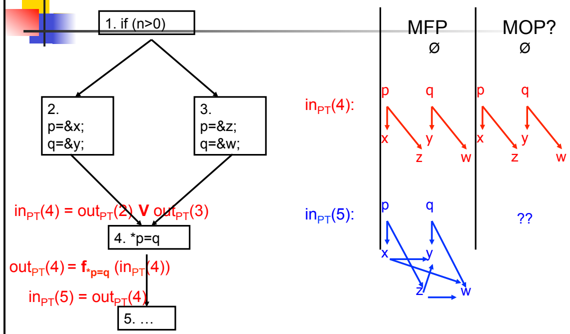
- Because of updates!



## Points-to Analysis is Not Distributive



## MFP vs. MOP for Points-to






## Andersen's Points-to Analysis

- Commonly attributed to Lars Andersen [1994]
- More approximation than our earlier formulation: don't ever "kill", thus, can maintain a single points-to graph
  - Straight-forward analysis formulation
    - Formulated in terms of subset constraints
    - Solvable by a version of the fixpoint iteration algorithm
  - Commonly referred to as a flow-insensitive, context-insensitive analysis



## Andersen's Points-to Analysis

**pts(p)** denotes the points-to set of **p**

(1) p = &a **{ a }** ⊑ **pts(p)**
(2) p = q **pts(q)** ⊑ **pts(p)**
(3) p = *q for each **x** in **pts(q)**. **pts(x)** ⊑ **pts(p)**
(4) *p = q for each **x** in **pts(p)**. **pts(q)** ⊑ **pts(x)**

Use worklist-like algorithm to compute least solution of these constraints



## Andersen's Points-to Analysis: Examples

Example 1:

p1 = &a
p2 = p1
*p2 = 1

Example 2:

p3 = &p1
p1 = &a
…
q = p3
r = *q
p1 = &b



## Outline of Today's Class

- One brief note on the Worklist algorithm
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis
- Analysis scope and approximations



## Analysis Scope

- **Intra**procedural analysis
  - Scope is the CFG of a single subroutine
  - Assumes no call and returns in routine, or models calls and returns
  - What we did so far
- **Inter**procedural analysis
  - Scope of analysis is the ICFG (Interprocedural CFG), which models flow of control between routines
  - A lot more on this to come!

## Analysis Scope

- **Whole-program** analysis
  - Assumes entry point "main"
  - Application code + standard libraries
    - Intricate interdependences, e.g., Android apps
- **Modular** analysis
  - Scope either a library without entry point
  - or application code with missing libraries
  - … or a library that depends on other (missing) libraries



## Approximations

- Dimensions of approximation
  - Transfer function space
  - Lattice
- Flow sensitivity
- Context sensitivity

(Somewhat poorly defined notions)



## Flow Sensitivity

- Flow-sensitive vs. flow-insensitive analysis
- Flow-sensitive analysis maintains the CFG and computes a solution per each node in CFG (i.e. each program point)
  - Standard dataflow analysis is flow-sensitive
- For large programs, maintaining CFG and solution per program point does not scale



## Flow Insensitivity

- Flow-insensitive analysis essentially discards CFG edges, computes a single solution S
  - E.g., Andersen's points-to analysis is flow-insensitive
- A "declarative" worklist-like algorithm:

```
S = 0
do {
  for each node j do
    S = fj(S) V S
} while S changes
```



## Flow Insensitivity

- A more "operational" worklist-like algorithm:

```
S = 0, W = { 1, 2, … n } /* all nodes */
while W ≠ Ø do {
  remove j from W
  S = fj(S) V S
  if S changed then
    W = W U { k | k is "successor" of j }
}
```

- Note that "successors" here does not refer to successor nodes in the CFG, but nodes **k** whose transfer function $\mathbf{f_k}$ may contribute to S as a result of the change by **j**

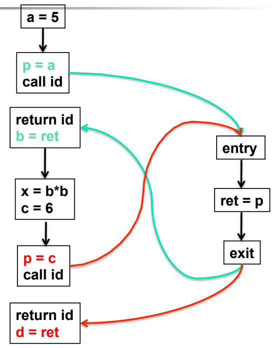


## Context Sensitivity

- Context-sensitive vs. context-insensitive
- So far, we did intraprocedural analysis
- Once we consider interprocedural analysis the issue of context-sensitive vs. context-insensitive comes up
- Context-insensitive analysis treats calls and returns as assignments
  - Can be flow-sensitive or flow-insensitive
- A lot more on context-sensitive analysis…

## Context Insensitivity

```
int id(int p) {
  return p;
}
    a = 5;
c1: b = id(a);
    x = b*b;
    c = 6;
c2: d = id(c);
```






## Your Homework

- A bunch of flow-insensitive, context-insensitive analyses for Java
  - All of RTA, XTA, PTA, etc.
  - Simple transfer functions
    - E.g., RTA gets rid of most nodes, processes just 2 kinds of nodes
  - Simple property space
  - Millions of lines of code in seconds