Lecture 13
Pointer Analysis

1. Datalog
2. Context-insensitive, flow-insensitive pointer analysis
3. Context sensitivity

Readings: Chapter 12

Goals of the Lecture

- Pointer analysis
  - Interprocedural, context-sensitive, flow-insensitive (Dataflow: intraprocedural, flow-sensitive)
- Power of languages and abstractions
- Elegant abstractions
  - Datalog: A deductive database (A database that can make deductions from stored data)
  - BDDs: Binary decision diagrams (Most cited CS papers for many years)

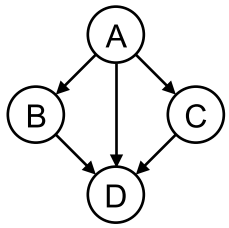
1. Why a Deductive Database?

- Pointer analysis produces “intermediate” results to be consumed in analysis.
- Allow query of specific subsets of results
- Analysis as queries
- Results of queries can be further queried in a uniform way

Datalog Basics

- p(X_1, X_2, … X_n)
  - p is a predicate
  - X_1, X_2, … X_n are terms such as variables or constants
- A predicate can be viewed as a relation

Example: Call graph edges Predicate vs. Relation



calls(A,B)
calls(A,C)
calls(A,D)
calls(B,D)
calls(C,D)

Predicates

- Calls (x,y): x calls y is true
- Ground atoms:
  predicates with constant arguments

Relations

- Calls (x,y) :
  x, y is in a "calls" relationship
- Extensional database:
  tuples representing facts

Datalog Programs: Set of Rules (Intensional DB)

- H :- B_1 & B_2 … & B_n
- LHS is true if RHS is true
  - Rules define the intensional database
- Example: Datalog program to compute call*
  - transitive closure of calls relation
  - calls*(x, y) if x calls y directly or indirectly
  - calls* (x, y) :- calls (x, y)
  - calls* (x, z) :- calls* (x, y) & calls* (y, z)
- Result:
  - set of ground atoms inferred by applying the rules until no new inferences can be made

Datalog vs. SQL

- SQL
  - Imperative programming:
    - join, union, projection, selection
  - Explicit iteration
- Datalog: logical database language
  - Declarative programming
  - Recursive definition: fixpoint computation
  - Negation can lead to oscillation
  - Stratified: separates rules into groups
    - Compute one group at a time
    - Can negate only the results from previous strata

Datalog vs. Prolog

- Syntactically a subset of Prolog
- No function variables e.g. b in a(b(x,y), c)
- Truly declarative:
  - Rule ordering does not affect program semantics
- Bottom-up evaluation
  - Stratified Datalog always terminates on a finite database

2. Flow-insensitive Points-to Analysis

- Alias analysis:
  - Can two pointers point to the same location?
  - *a, *(a+8)
- Points-to analysis:
  - What objects does each pointer points to?
  - Two pointers cannot be aliased
    if they must point to different objects

How to Name Objects?

- Objects are dynamically allocated
- Use finite names to refer to unbounded # objects
- 1 scheme: Name an object by its allocation site

```
main () {            f () {
    p = f();             A: a = new O ();
    q = f();             B: b = new O ();
}                            return a;
                     }
```

Points-To Analysis for Java

- Variables ($v \in V$):
  - local variables in the program
- Heap-allocated objects ($h \in H$)
  - has a set of fields ($f \in F$)
  - named by allocation site

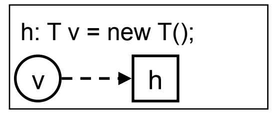
Program Abstraction

- Allocations h: v = new c
- Store $v_1.f = v_2$
- Loads $v_2 = v_1.f$
- Moves, arguments: $v_1 = v_2$
- Assume: a (conservative) call graph is known a priori
  - Call: formal = actual
  - Return: actual = return value

Pointer Analysis Rules

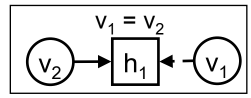
Object creation

$pts(v, h)$:- "h: T v = new T()".



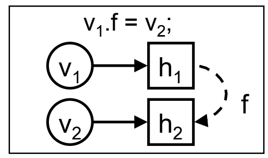
Assignment

$pts(v_1, h_1)$:- "$v_1 = v_2$" & $pts(v_2, h_1)$.



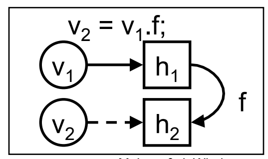
Store

$hpts(h_1, f, h_2)$:- "$v_1.f = v_2$" &
$pts(v_1, h_1)$ & $pts(v_2, h_2)$.



Load

$pts(v_2, h_2)$:- "$v_2 = v_1.f$" &
$pts(v_1, h_1)$ & $hpts(h_1, f, h_2)$.



Pointer Alias Analysis

- Specified by a few Datalog rules
  - Creation sites
  - Assignments
  - Stores
  - Loads
- Apply rules until they converge

Example program

```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```


Pointer Analysis in Datalog

Domains
- V = variables
- H = heap objects
- F = fields

EDB (input) relations
- vP_0 (v:V, h:H) : object allocation sites
- assign(v_1:V, v_2:V) : assignment instructions ($v_1 = v_2$;) and parameter passing
- store (v_1:V, f:F, v_2:V) : store instructions ($v_1.f = v_2$;)
- load (v_1:V, f:F, v_2:V) : load instructions ($v_2 = v_1.f$;)

IDB (computed) relations
- vP (v:V ,h:H) : variable points-to relation (v can point to object h)
- hP (h_1:H, f:F, h_2:H) : heap points-to relation (object h_1 field f can point to h_2)

Rules
- vP (v, h) :- vP_0 (v, h).
- vP (v_1, h) :- assign (v_1, v_2), vP (v_2, h).
- hP (h_1, f, h_2) :- store (v_1, f, v_2), vP (v_1, h_1), vP (v_2, h_2).
- vP (v_2, h_2) :- load (v_1, f, v_2), vP (v_1, h_1), hP(h_1, f, h_2).

Step 1: Assign numbers to elements in domain

```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```

Domains

V
- 'x' : 0
- 'y' : 1
- 'z' : 2
- 'a' : 3
- 'b' : 4
- 'c' : 5
- 'd' : 6

H
- 'main@1' : 0
- 'main@2' : 1
- 'main@3' : 2

F
- 'f' : 0

Step 2: Extract initial relations (EDB) from program

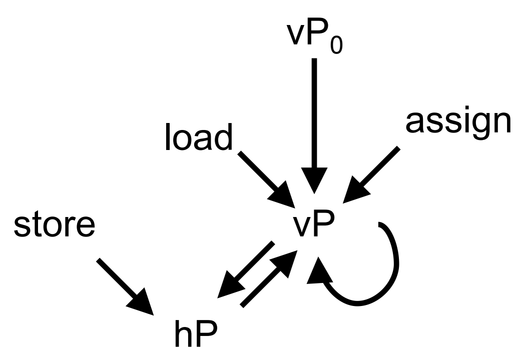
```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```

vP_0('x', 'main@1').
vP_0('y', 'main@2').
vP_0('z', 'main@3').
assign('a','x').
assign('b','y').
assign('c','z').
assign('d','x').
load('z','f','q').
assign('c','a').
assign('d','b').
store('c','f','d').

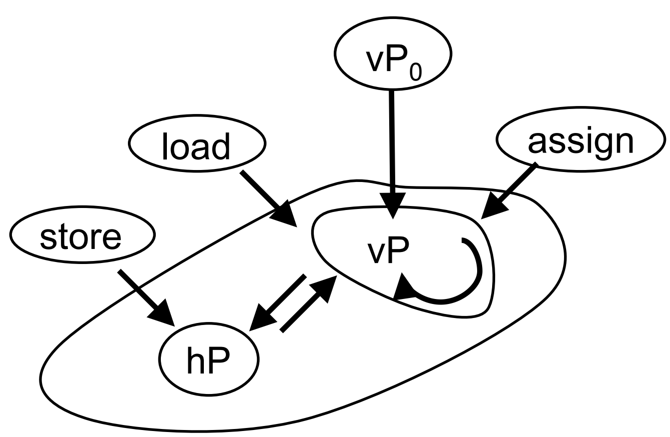
Step 3: Generate Predicate Dependency Graph

Rules

$vP(v,h) \text{ :- } vP_0(v,h).$
$vP(v_1,h) \text{ :- } assign(v_1,v_2),\ vP(v_2,h).$
$hP(h_1,f,h_2) \text{ :- } store(v_1,f,v_2),\ vP(v_1,h_1),\ vP(v_2,h_2).$
$vP(v_2,h_2) \text{ :- } load(v_1,f,v_2),\ vP(v_1,h_1),\ hP(h_1,f,h_2).$



Step 4: Determine Iteration Order



Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a','x').
assign('b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

hP

Advanced Compilers

M. Lam & J. Whaley

Step 5: Apply rules until convergence

Rules

→ $vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a','x').
assign('b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').

hP

Advanced Compilers

M. Lam & J. Whaley

Step 5: Apply rules until convergence

Rules

- vP(v,h) :- vP_0(v,h).
- vP(v_1,h) :- assign(v_1,v_2), vP(v_2,h).
- hP(h_1,f,h_2) :- store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2).
- vP(v_2,h_2) :- load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2).

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').

hP

Step 5: Apply rules until convergence

Rules

- vP(v,h) :- vP_0(v,h).
- vP(v_1,h) :- assign(v_1,v_2), vP(v_2,h).
- hP(h_1,f,h_2) :- store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2).
- vP(v_2,h_2) :- load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2).

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').

hP

Step 5: Apply rules until convergence

Rules

- vP(v,h) :- vP_0(v,h).
- vP(v_1,h) :- assign(v_1,v_2), vP(v_2,h).
- → hP(h_1,f,h_2) :- store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2).
- vP(v_2,h_2) :- load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2).

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').

hP

hP('main@1','f','main@1').
hP('main@1','f','main@2').
hP('main@3','f','main@1').
hP('main@3','f','main@2').

Step 5: Apply rules until convergence

Rules

- vP(v,h) :- vP_0(v,h).
- vP(v_1,h) :- assign(v_1,v_2), vP(v_2,h).
- hP(h_1,f,h_2) :- store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2).
- → vP(v_2,h_2) :- load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2).

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

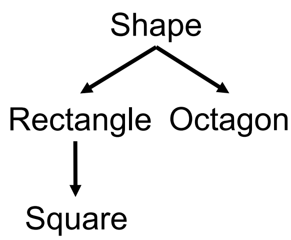
vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').
vP('q','main@1').
vP('q','main@2').

hP

hP('main@1','f','main@1').
hP('main@1','f','main@2').
hP('main@3','f','main@1').
hP('main@3','f','main@2').

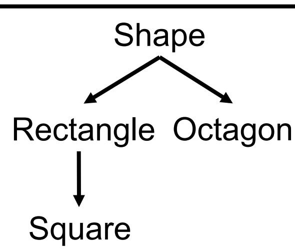
Virtual Method Invocation




```
void draw (shape s) {
      int i = s.lines();
      …
  }
```

- Class hierarchy analysis cha (t, n, m)
  - Given an invocation v.n (…), if v points to object of type t, then m is the method invoked
  - t's first superclass that defines n

Virtual Method Invocation




```
void draw (shape s) {
      int i = s.lines();
      …
  }
```


- Class hierarchy analysis cha (t, n, m)
  - Simple analysis: can determine the type if the program only allocates one type of objects.

Pointer Analysis Can Improve Call Graphs

Discover points-to results and methods invoked on the fly

invokes (s, m): statement s calls method m

hType (h, t): h has type t

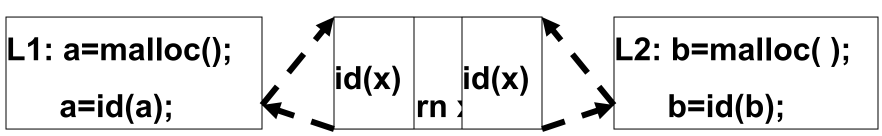
invokes (s, m) :- "s: v.n (…)" & pts (v, h) & hType (h, t) & cha (t, n, m)

actual (s, i, v): v is the ith actual parameter in call site s.

formal (m, i, v): v is the ith formal parameter declared in method m.

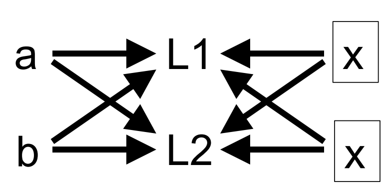
pts(v, h) :- invokes (s, m) & formal (m, i, v) & actual (s, i, w) & pts (w, h)

3. Context-Sensitive Pointer Analysis

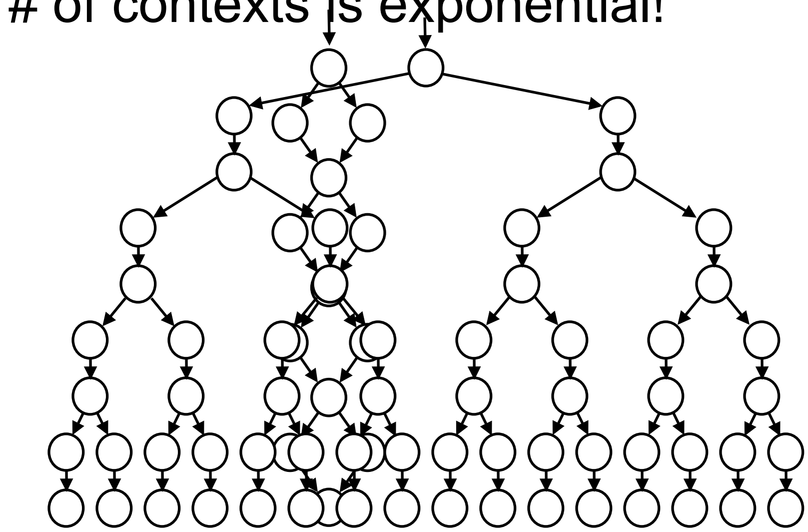


context-sensitive

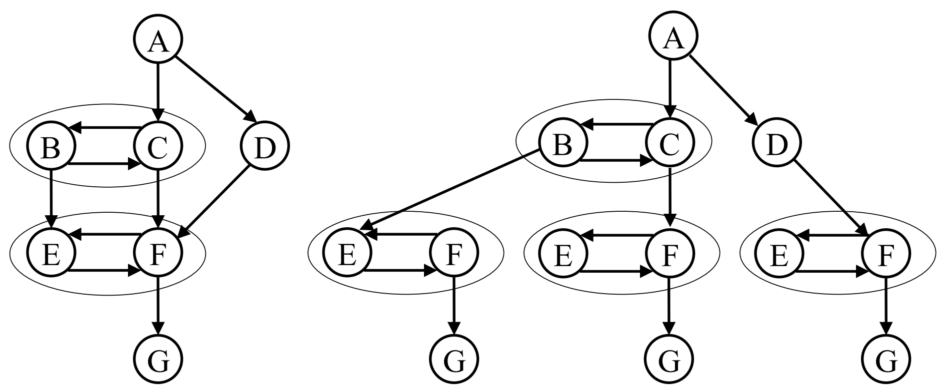
context-insensitive



Even without recursion, # of contexts is exponential!

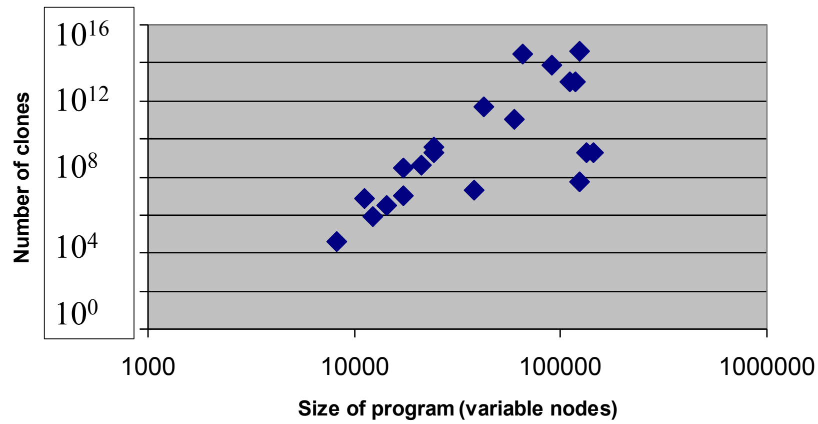


Recursion



Top 20 Sourceforge Java Apps

Number of Clones



Cloning-Based Algorithm

- Apply the context-insensitive algorithm to the program to discover the call graph
- Find strongly connected components
- Create a “clone” for every context
- Apply the context-insensitive algorithm to cloned call graph
- Lots of redundancy in result
- Exploit redundancy by clever use of BDDs (binary decision diagrams)

Whaley&Lam, PLDI 2004 (best paper award)

Automatic Analysis Generation

Programmer:
Security analysis
in 10 lines

PQL

Compiler Writer:
Ptr analysis in 10 lines

Datalog

bddbddb
(**BDD**-**b**ased
deductive **d**ata**b**ase)
with
Active Machine Learning

1000s of lines
1 year tuning

BDD operations

BDD: 10,000s-lines library