Pointer Analysis

Static Program Analysis

Christian Hammer

15 Dec 2014

SPONSORED BY THE

Federal Ministry of Education and Research

max planck institut informatik

German Research Center for Artificial Intelligence

Max Planck Institute for Software Systems

Pointer Analysis (Points-to Analysis)

CISPA Center for IT-Security, Privacy and Accountability

- Which objects may each pointer in a program point to at run-time
- Required by compiler analyses and optimizations, program understanding and verification tools
- e.g. resolving dynamic binding

- ```
  class A { String foo() { return "A"; }}
  class B extends A {
   @override foo() {
    return "B";
   }
  }
  ...
  A x;
  ```
- x.foo(); <- which method is invoked here? What is the return value?
  ```

Alias Analysis

- When do two pointers (references) point to the same object?

- Can be introduced due to assignments
  a = new A()
  b = a
- Also field assignments/reads
  o.f = a
  b = o.f
- Or via parameter passing
  foo(a, a)
  foo(Object x, Object y) {...}

Dimensions/Challenges in Pointer Analysis

- Intraprocedural / interprocedural
- Flow-sensitive / flow-insensitive
- Context-sensitive / context-insensitive
- Definiteness
  - May versus must
- Heap modeling
- Representation

Flow-Sensitivity

- Flow-insensitive: Compute one solution for the whole program
  Which objects may this reference ever point to?
- Flow-sensitive: Compute a solution for each point in the program
  Which objects may this reference point to before this statement?
- Flow-sensitive analysis has been considered too expensive for practical use
  - in particular for realistic programs
  - depending on the language features can be NP-hard
  - e.g. with multiple level pointers
  - local flow-sensitivity can be achieved via SSA-form

Context-Sensitivity

- Hard in practice as no good summary-based analysis available
- Inlining with k-limited calling contexts used widely

- Scaling to hundreds of thousands LOC using
  - BDDs (Binary Decision Diagrams)
  - Datalog
  - Databases with fast Datalog interpreters

- Alternative: Object-sensitive Analysis
  Target object used as calling context

Definiteness

- May-aliasing required for e.g. reaching definitions (gen-sets)

  x.f = a
  ...
  b = y.f
- if x and y are may-aliased b is potentially dependent on the value of a

- must-aliasing required to determine kill-sets

  x.f = a
  ...
  y.f = b
  ...
  c = x.f
- if x and y are must-aliases the first definition of field f is overwritten (killed) by the second definition of field f

Abstract Semantics

- Need to model the “flow” of objects into references and fields
- x = new A
  - x ⊇ {A}, meaning loc(A) ∈ pt(x)
- y = x
  - y ⊇ x (Andersen), y = x (Steensgaard)
- a.f = x, y = a.f
  - ??? (depends on value of reference a)

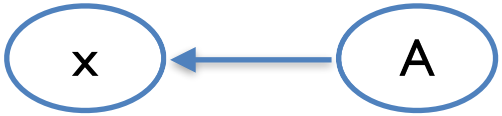
- Steensgaard scales better, but less precise
  - Union-Find-Algorithm O(nα(n)) (almost linear)
  - Disadvantage: Leads to typing issues for e.g. Java programs
  - Not used much for strongly typed languages and virtual binding resolution

Problem

- We need an abstraction for dynamic object creation sites:

```
for (…) {
  x = new A()
  …
}
x.foo()
```

- Insight: exact instance of class A not required, type is enough here!
- Solution: One equivalence class for all objects created at the allocation site
- $x \supseteq \{A\}$



Analysis Rules

	Allocation	Assignment	Field store	Field load
Instruction	$l := new\ C$	$l_i := l_j$	$l_i.f := l_j$	$l_i := l_j.f$
Edge	$new\ C \rightarrow l$	$l_j \rightarrow l_i$	$l_j \rightarrow l_i.f$	$l_j.f \rightarrow l_i$
Rules	$\dfrac{new_i \rightarrow l}{new_i \in pt(l)}$	$\dfrac{l_j \rightarrow l_i \quad new_i \in pt(l_j)}{new_i \in pt(l_i)}$	$\dfrac{l_j \rightarrow l_i.f \quad new_i \in pt(l_j) \quad new_j \in pt(l_i)}{new_i \in pt(new_j.f)}$	$\dfrac{l_j.f \rightarrow l_i \quad new_i \in pt(l_j) \quad new_j \in pt(new_i.f)}{new_j \in pt(l_i)}$

Example


- ```
  String[][] st = new String[3][4];
  s = "Something";
  st[1][0] = s;
  s = "Hello";
  s = st[1][0];
  int i6=0;
  String s6 = s;
  if (i6==0) s6 = "something else";
  ```

- Which strings can s6 point to in the end?
  ```

Iterative Algorithm

initialize sets according to allocation edges
repeat
    propagate sets along each assignment edge $p \longrightarrow q$
    for each load edge $p.f \longrightarrow q$ **do**
        for each $a \in pt(p)$ **do**
            propagate sets $pt(a.f) \longrightarrow pt(q)$
    for each store edge $p \longrightarrow q.f$ **do**
        for each $a \in pt(q)$ **do**
            propagate sets $pt(p) \longrightarrow pt(a.f)$
until no changes

Worklist Algorithm (more efficient)

```
for each allocation edge o1 → p do
   pt(p) ∪= {o1}
   add p to worklist
repeat
   repeat
      remove first node p from worklist
      propagate sets along each assignment edge p → q,
      adding q to worklist whenever pt(q) changes
      for each store edge q → r.f where p = q or p = r
      do
         for each a ∈ pt(r) do
            propagate sets pt(q) → pt(a.f)
      for each load edge p.f → q do
         for each a ∈ pt(p) do
            propagate sets pt(a.f) → q
            add q to worklist if pt(q) changed
   until worklist is empty
   for each store edge q → r.f do
      for each a ∈ pt(r) do
         propagate sets pt(q) → pt(a.f)
   for each load edge p.f → q do
      for each a ∈ pt(p) do
         propagate sets pt(a.f) → q
         add q to worklist if pt(q) changed
until worklist is empty
```

Interprocedural Analysis

- Often simple inlining (context-insensitive analysis)
- Contexts for method calls only, sometimes even for allocation sites
- Special case: Objects-sensitive Pointer Analysis
  - context is target object (k-limiting k=1)
  - good precision in practice with moderate overhead
  - in particular, does not merge different container objects (Lists, Vectors,…)

- Vector v1 = new Vector(), v2 = new Vector()
  v1.add("Hello")
  v2.add("World")
  x = v1.get(1)
  - pt(x) = {"Hello"} only
- Call graph construction interleaved with pointer analysis
  - precision of pointer analysis removes infeasible dynamic dispatch
  - further improves the pointer analysis results

Example

- class A { String foo() { return “A” }}
  class B extends A {String foo() { return “B” }}
  main() { A a = new A(); A b = new B(); x = b.foo(); }
- A.foo returns “A”, B.foo returns “B”
- b has type A
- Class Hierarchy Analysis of this program determines that b.foo() could be dispatched to either A.foo or B.foo
- But only new B() is in pt(b)
  - thus the virtual call to b.foo() can only be dispatched to B.foo
- pt(x) = {“B”} and not {“A,B”}

Datalog

- Logical query language (from databases)
- Non-recursive rules are equivalent to the core relational algebra
- Implementation for Program Analysis: bddbddb (BDD-Based deductive DataBase)
- New startup: database with datalog support

Rule = Atom :- Literal, Literal, …, Literal.

Make this atom true…

Head

…for each assignment of values to variables that make all these true

Body

Literal = Atom | !Atom

Atom = Predicate(<list of arguments>)

Datalog Program

Rule =	Atom :- Literal, Literal, …, Literal.
Literal =	Atom | !Atom
Atom =	Predicate(<list of arguments>)

- A datalog program is a collection of rules
- Predicates can be external or internal
- EDB: Extensional Database (external information, e.g. stored in a table, from the CFG, prior analysis, …)
- IDB: Intensional Database (relation only defined by rules)
- Either IDB or EDB, never both
- No EDB in heads

Example: Reaching Definitions

- Reach(d,x,j) :- Reach(d,x,i), StatementAt(i,s), !Assign(s,x), Succ(i,j).

Reach(d,x,j)	definition d of variable x reaches node j
Reach(d,x,i)	definition d of variable x reaches node i
StatementAt(i,s)	statement s in node i
Assign(s,x)	x is assigned new value in s
Succ(i,j)	node j is a successor of node i

- Reach(s,x,j) :- StatementAt(i,s), Assign(s,x), Succ(i,j).

- New assignment always reaches successor

Negation vs. Fixed Points

- Negation tricky in recursive predicates:

P(X) = E(X), !P(X).

- Given E(X) is true, is P(X) true?
- Iteration 1: Yes, 2: No, 3: Yes, …

- Semantics of Negation:
  - No Negation allowed [Ullman 1988]
  - Stratified Datalog [Chandra 1985]
  - Well-founded Semantics [Van Gelder 1991]

Evaluation of Negated Predicates

- Requirement: Evaluation order < over IDB Predicates

$$P \text{ :- } \ldots, !Q, \ldots . \quad \Rightarrow \; Q \text{ is EDB} \vee Q < P$$

- Intuition: Negations can be evaluated before the formula that they are used in.

- Iterative Evaluation algorithm:
  - Start with EDB predicates and leave all IDB predicates un-evaluated
  - Iteratively evaluate IDB rules in order of strata (low to high) (order between IDB of the same stratum does not matter)
  - End when no change to IDB rules (fixed point)

(Simplified) Intra-Procedural Pointer analysis: Relations

- CFG etc. are modeled as EDB

- Input vP0 (variable: V, heap H)
  // vP0(v,h) true iff the program places a reference to heap obj h in v
- Input fldst (bytecode: B, base: V, field: F, source: V)
  // fldst(b,v1,f,v2) means that bytecode b executes v1.f = v2
- Input fldld (bytecode: B, dest: V, field: F, base: V)
  // fldld(b,v1,f,v2) means that bytecode b executes v1 = v2.f
- Input assign (dest: V, source: V)
  // assign(v1,v2) true iff the program contains assignment v1=v2
- Output vP (variable: V, heap: H)
  // vP(v,h) iff v points to heap obj h at any point during execution
- Output hP (base: H, field: F, target: H)
  // hP(h1,f,h2) true iff heap object h1.f may point to h2

(Simplified) Intra-Procedural Pointer Analysis: Rules

- Input vP0 (variable: V, heap H)
  Input fldst (bytecode: B, base: V, field: F, source: V)
  Input fldld (bytecode: B, dest: V, field: F, base: V)
  Input assign (dest: V, source: V)
  Output vP (variable: V, heap: H)
  Output hP (base: H, field: F, target: H)

```
vP(v,h)       :-    vP0(v,h).    //initialize points-to relation
vP(v1, h)     :-    assign(v1,v2), vP(v2,h). //transitive closure (assignments)
hP(h1,f,h2)   :-    fldst(_, v1, f, v2), vP(v1,h1), vP(v2,h2).
        //stores to fields, e.g. o.f = x;
vP(v2,h2)     :-    fldld(_,v1,f,v2),   vP(v1,h1), hP(h1,f,h2).
        //loads of fields, e.g. x = o.f;
```

(Context-Insensitive) Pointer Analysis in BDDBDDB

CISPA Center for IT-Security, Privacy and Accountability

- # Type 0 is Object
  aT(t0,t1) :- aT0(t0,t1).
  aT(t0,t2) :- aT(t0,t1), aT(t1,t2).
  aT(0,t1) :- aT0(t1,_).
  vT(v0,t0) :- vT0(v0,t0).
  vT(v0,t0) :- vT0(v0,t0).
  vT(v0,0) :- !vT0(v0,_).

- vP(v,h) :- vP0(v,h).
  IE(i,m) :- IE0(i,m).
  vPfilter(v,h) :- vT(v,tv), aT(tv,th), hT(h,th).
  vP(v1,h) :- A(v1,v2), vP(v2,h), vPfilter(v1,h).
  hP(h1,f,h2) :- S(v1,f,v2), vP(v1,h1), vP(v2,h2).
  vP(v2,h2) :- L(v1,f,v2), vP(v1,h1), hP(h1,f,h2), vPfilter(v2,h2). split
  A(v1,v2) :- formal(m,z,v1), IE(i,m), actual(i,z,v2).
  A(v2,v1) :- Mret(m,v1), IE(i,m), Iret(i,v2).
  A(v2,v1) :- Mthr(m,v1), IE(i,m), Ithr(i,v2).

vP0 (v:V0, h:H0) input
vP (v:V0, h:H0) output printtuples
A (dest:V0, source:V1) input
hP0 (ha:H0, field:F0, hb:H1) input
hP (ha:H0, field:F0, hb:H1) output
S (base:V0, field:F0, src:V1) input
L (base:V0, field:F0, dest:V1) input
vT (var:V0, type:T0)
hT (heap:H0, type:T1) input
aT (type:T0, type:T1)
vT0 (var:V0, type:T1) input
aT0 (type:T0, type:T1) input
vPfilter (v:V0, h:H0)
cha (type:T1, name:N0, method:M0) input
actual (invoke:I0, num:Z0, actualparam:V1) input
formal (method:M0, num:Z0, formalparam:V0) input
Mret (method:M0, v:V1) input
Mthr (method:M0, v:V1) input
Iret (invoke:I0, v:V0) input
Ithr (invoke:I0, v:V0) input
mI (method:M0, invoke:I0, name:N0) input
IE0 (invoke:I0, target:M0) input
IE (invoke:I0, target:M0) output printtuples