(Pointer Analysis)

Static Program Analysis

Christian Hammer

15 Dec 2014
Pointer Analysis (Points-to Analysis)

- 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

```java
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
  
  \[
  a = \text{new } A() \\
  b = a
  \]

- Also field assignments_reads
  
  \[
  o.f = a \\
  b = o.f
  \]

- Or via parameter passing
  
  \[
  \text{foo}(a, a) \\
  \text{foo}(\text{Object } x, \text{Object } y) \{...\} 
  \]
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)

  \[ x.f = a \]
  
  \[ \ldots \]
  
  \[ b = y.f \]

  - if \( x \) and \( y \) are may-aliased \( b \) is potentially dependent on the value of \( a \)

- must-aliasing required to determine kill-sets

  \[ x.f = a \]
  
  \[ \ldots \]
  
  \[ y.f = b \]
  
  \[ \ldots \]
  
  \[ c = x.f \]

  - 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
  
  - \( x = \text{new} \ A \)
    
    - \( x \supseteq \{A\} \), meaning \( \text{loc}(A) \in \text{pt}(x) \)
  
- \( y = x \)
  
  - \( y \supseteq x \) (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 \( O(n\alpha(n)) \) (almost linear)
  
  - Disadvantage: Leads to typing issues for e.g. Java programs
  
  - Not used much for strongly typed languages and virtual binding resolution
We need an abstraction for dynamic object creation sites:

```java
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Allocation</th>
<th>Assignment</th>
<th>Field store</th>
<th>Field load</th>
</tr>
</thead>
<tbody>
<tr>
<td>new C → l</td>
<td>l := new C</td>
<td>l_i := l_j</td>
<td>l_i.f := l_j</td>
<td>l_i := l_j.f</td>
</tr>
</tbody>
</table>

#### Rules

<table>
<thead>
<tr>
<th>Rules</th>
<th>Allocation</th>
<th>Assignment</th>
<th>Field store</th>
<th>Field load</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_i → l</td>
<td>l_j → l_i / new_i ∈ pt(l_j)</td>
<td>l_j → l_i.f / new_i ∈ pt(l_j)</td>
<td>l_j → l_i.f / new_i ∈ pt(l_j)</td>
<td>l_j → l_i.f / new_i ∈ pt(l_j)</td>
</tr>
<tr>
<td></td>
<td>new_i ∈ pt(l)</td>
<td>new_j ∈ pt(l_i)</td>
<td>new_j ∈ pt(new_i.f)</td>
<td>new_j ∈ pt(l_i)</td>
</tr>
</tbody>
</table>
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

1: initialize sets according to allocation edges
2: repeat
3: propagate sets along each assignment edge $p \rightarrow q$
4: for each load edge $p.f \rightarrow q$ do
5: for each $a \in \text{pt}(p)$ do
6: propagate sets $\text{pt}(a.f) \rightarrow \text{pt}(q)$
7: for each store edge $p \rightarrow q.f$ do
8: for each $a \in \text{pt}(q)$ do
9: propagate sets $\text{pt}(p) \rightarrow \text{pt}(a.f)$
10: until no changes
Worklist Algorithm (more efficient)

1: for each allocation edge $o_1 \rightarrow p$ do
2: \hspace{1em} $pt(p) \cup= \{o_1\}$
3: \hspace{1em} add $p$ to worklist
4: repeat
5: \hspace{1em} repeat
6: \hspace{2em} remove first node $p$ from worklist
7: \hspace{2em} propagate sets along each assignment edge $p \rightarrow q$, adding $q$ to worklist whenever $pt(q)$ changes
8: \hspace{2em} for each store edge $q \rightarrow r.f$ where $p = q$ or $p = r$ do
9: \hspace{3em} for each $a \in pt(r)$ do
10: \hspace{4em} propagate sets $pt(q) \rightarrow pt(a.f)$
11: \hspace{2em} for each load edge $p.f \rightarrow q$ do
12: \hspace{3em} for each $a \in pt(p)$ do
13: \hspace{4em} propagate sets $pt(a.f) \rightarrow q$
14: \hspace{4em} add $q$ to worklist if $pt(q)$ changed
15: until worklist is empty
16: for each store edge $q \rightarrow r.f$ do
17: \hspace{1em} for each $a \in pt(r)$ do
18: \hspace{2em} propagate sets $pt(q) \rightarrow pt(a.f)$
19: \hspace{1em} for each load edge $p.f \rightarrow q$ do
20: \hspace{2em} for each $a \in pt(p)$ do
21: \hspace{3em} propagate sets $pt(a.f) \rightarrow q$
22: \hspace{3em} add $q$ to worklist if $pt(q)$ changed
23: 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”}
Logical query language (from databases)
- Non-recursive rules are equivalent to the core relational algebra
- Implementation for Program Analysis: bdddbdb (BDD-Based deductive DataBase)
- New startup: database with datalog support

Datalog

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

Make this atom true...
...for each assignment of values to variables that make all these true

Literal = Atom | !Atom
Atom = Predicate(<list of arguments>)
Datalog Program

- 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

Rule = Atom :- Literal, Literal, ..., Literal.
Literal = Atom | !Atom
Atom = Predicate(<list of arguments>)
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:
  
  \[ P(X) = E(X), \neg P(X). \]

- 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

$$P : - \ldots, !Q, \ldots, \Rightarrow Q \text{ is EDB } \lor Q < P$$

- 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)
  \[// \ vP0(v,h) \text{ true iff the program places a reference to heap obj } h \text{ in } v\]

- **Input** $fldst$ (bytecode: B, base: V, field: F, source: V)
  \[// \ fldst(b,v1,f,v2) \text{ means that bytecode } b \text{ executes } v1.f = v2\]

- **Input** $fldld$ (bytecode: B, dest: V, field: F, base: V)
  \[// \ fldld(b,v1,f,v2) \text{ means that bytecode } b \text{ executes } v1 = v2.f\]

- **Input** $assign$ (dest: V, source: V)
  \[// \ assign(v1,v2) \text{ true iff the program contains assignment } v1=v2\]

- **Output** $vP$ (variable: V, heap: H)
  \[// \ vP(v,h) \text{ iff } v \text{ points to heap obj } h \text{ at any point during execution}\]

- **Output** $hP$ (base: H, field: F, target: H)
  \[// \ hP(h1,f,h2) \text{ true iff heap object } h1.f \text{ may point to } h2\]
(Simplified) Intra-Procedural Pointer Analysis: Rules

- **Input** $vP_0$ (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)

\[
vP(v, h) \quad :- \quad vP_0(v, h). \quad //\text{initialize points-to relation}
\]

\[
vP(v1, h) \quad :- \quad assign(v1, v2), vP(v2, h). \quad //\text{transitive closure (assignments)}
\]

\[
hP(h1, f, h2) \quad :- \quad fldst(\_\_\_, v1, f, v2), vP(v1, h1), vP(v2, h2).
\]

\[
\quad \quad //\text{stores to fields, e.g. } o.f = x;
\]

\[
vP(v2, h2) \quad :- \quad fldld(\_\_\_, v1, f, v2), \quad vP(v1, h1), \quad hP(h1, f, h2).
\]

\[
\quad \quad //\text{loads of fields, e.g. } x = o.f;
\]
(Context-Insensitive) Pointer Analysis in BDDBDB

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

- \[ vP(v,h) :- vP0(v,h). \]

- \[ IE(i,m) :- IE0(i,m). \]

- \[ vPfilter(v,h) :- vT(v, tv), aT(tv, th), hT(h, th). \]

- \[ vP(v1, h) :- A(v1, v2), vP(v2, h), vPfilter(v1, h). \]

- \[ hP(h1, f, h2) :- S(v1, f, v2), vP(v1, h1), vP(v2, h2). \]

- \[ vP(v2, h2) :- L(v1, f, v2), vP(v1, h1), hP(h1, f, h2), vPfilter(v2, h2). \]

- \[ A(v1, v2) :- formal(m, z, v1), IE(i, m), actual(i, z, v2). \]

- \[ A(v2, v1) :- Mret(m, v1), IE(i, m), Iret(i, v2). \]

- \[ A(v2, v1) :- Mthr(m, v1), IE(i, m), Ithr(i, v2). \]