

AAA615: Formal Methods

Lecture 8 — Pointer Analysis

Hakjoo Oh
2017 Fall

Topics

- Pointer analysis
- Constraint-based analysis
- Interprocedural analysis
- Analysis of higher-order programs
- Context-sensitivity

Motivating Example

Reasoning about any real programs needs pointer reasoning: e.g.,

```
x = 1;
```

```
y = 2;
```

```
*p = 3;
```

```
*q = 4;
```

What is the value of $x + y$ after the last statement?

- $p = \&x$ and $p = \&y$:
- $p = \&x$ and $p \neq \&y$:
- $p \neq \&x$ and $p = \&y$:
- $p \neq \&x$ and $p \neq \&y$:

Pointer Analysis

- Static program analysis that computes the set of memory locations (objects) that a pointer variable may point to at runtime.
- One of the most important static analyses: all interesting questions on program reasoning eventually need pointer analysis.
 - ▶ E.g., control-flows, data-flows, types, information-flows, etc

Allocation-Site Abstraction

Memory locations are unbounded. Consider the program:

```
Object id (Object p) { return p; }  
void f() {  
    Object x = new A()    // l1  
    Object y = id(x);  
}  
void g() {  
    Object a = new B();    // l2  
    Object b = id(a);  
}  
void main () { while (...) { f(); g(); } }
```

- In program execution, new objects are allocated repeatedly.
- In pointer analysis, objects get abstracted to their allocation sites.
- Thus, a pointer analysis would produce the result:

$$x \mapsto \{l_1\}, y \mapsto \{l_1\}, a \mapsto \{l_2\}, b \mapsto \{l_2\}, p \mapsto \{l_1, l_2\}$$

Pointer Analysis in Datalog

- Pointer analysis is expressed as subset constraints. The analysis is to compute the smallest solution of the constraints. E.g.,
a = new A();
b = a;
- We use the Datalog language to express such constraints.
- Datalog is a declarative logic programming language, which has application in many fields, e.g., database, information extraction, networking, program analysis, security, etc.

Syntax of Datalog

- A Datalog program is a sequence of constraints:

$$P ::= \bar{c}$$

- A constraint consists of a head of a literal and a body of a list of literals:

$$c ::= l :- \bar{l}$$

A constraint represents a horn clause (a disjunction of literals with at most one positive, unnegated, literal):

$$l \vee \neg l_1 \vee \neg l_2 \vee \dots \vee l_n \iff l \leftarrow l_1 \wedge l_2 \wedge \dots \wedge l_n$$

- A literal is a relation with arguments:

$$l ::= r(\bar{a})$$

where an argument is either a variable or constant.

Example

```
parent(bill, mary).  
parent(mary, john).  
ancestor(X,Y) :- parent(X,Y).  
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```


Semantics of Datalog

- A Datalog program denotes a set of ground literals:

$$\llbracket P \rrbracket \in \wp(G)$$

where G is the set of ground literals (literals without variables).

- A Datalog rule $l :- l_1, \dots, l_n$ denotes the function:

$$f_l :- l_1, \dots, l_n(X) = \{\sigma(l_0) \mid \sigma(l_k) \in X \text{ for } 1 \leq k \leq n\}$$

where σ is a variable substitution.

- The semantics of P is defined as the least fixed point of F_P :

$$\llbracket P \rrbracket = \text{lfp} F_P \quad \text{where} \quad F_P(X) = X \cup \bigcup_{c \in P} f_c(X)$$

- The semantics is monotone:

$$P_1 \subseteq P_2 \implies \llbracket P_1 \rrbracket \subseteq \llbracket P_2 \rrbracket$$

Programs as Relations

Without procedures, a program can be represented by a set of the relations:

$ALLOC (var : V, heap : H)$
 $MOVE (to : V, from : V)$
 $LOAD(to : V, base : V, fld : F)$
 $STORE(base : V, fld : F, from : V)$

domains:

- V is a set of program variables
- H is a set of heap abstractions (i.e. allocation sites)
- F is a set of fields

```
a = new A();  
b = new B();  
c = a;  
a.f = b;  
d = c.f;
```

Simple Pointer Analysis

Pointer analysis computes the set of points-to relations:

$$\text{VARPOINTS_TO}(var : V, heap : H)$$
$$\text{FLDPOINTS_TO}(baseH : H, fld : F, heap : H)$$

Analysis rules:

$$\text{VARPOINTS_TO}(var, heap) \leftarrow \text{ALLOC}(var, heap)$$
$$\begin{aligned} \text{VARPOINTS_TO}(to, heap) \leftarrow \\ \text{MOVE}(to, from), \text{VARPOINTS_TO}(from, heap) \end{aligned}$$
$$\begin{aligned} \text{FLDPOINTS_TO}(baseH, fld, heap) \leftarrow \\ \text{STORE}(base, fld, from), \text{VARPOINTS_TO}(from, heap), \\ \text{VARPOINTS_TO}(base, baseH). \end{aligned}$$

Exercise) Define the rule for LOAD.

Interprocedural Analysis

Domains:

- V is a set of program variables
- H is a set of heap abstractions (i.e. allocation sites)
- F is a set of fields
- M is a set of method identifiers
- S is a set of method signatures (including name, type signature)
- I is a set of instructions
- T is a set of class types
- \mathbb{N} is the set of natural numbers

Interprocedural Analysis (First-Order)

- Input relations:

ALLOC (*var* : V , *heap* : H , *inMeth* : M)

MOVE (*to* : V , *from* : V)

LOAD(*to* : V , *base* : V , *fld* : F)

STORE(*base* : V , *fld* : F , *from* : V)

CALLGRAPH(*invo* : I , *meth* : M)

REACHABLE(*meth* : M)

FORMALARG(*meth* : M , *i* : \mathbb{N} , *arg* : V)

ACTUALARG(*invo* : I , *i* : \mathbb{N} , *arg* : V)

FORMALRETURN(*meth* : M , *ret* : V)

ACTUALRETURN(*invo* : I , *var* : V)

- Output relations:

VARPOINTSTO (*var* : V , *heap* : H)

FLDPOINTSTO (*baseH* : H , *fld* : F , *heap* : H)

INTERPROCASSIGN (*to* : V , *from* : V)

Analysis Rules

$\text{VARPOINTS_TO}(var, heap) \leftarrow$
 $\text{REACHABLE}(meth), \text{ALLOC}(var, heap, meth)$

$\text{VARPOINTS_TO}(to, heap) \leftarrow$
 $\text{MOVE}(to, from), \text{VARPOINTS_TO}(from, heap)$

$\text{FLDPOINTS_TO}(baseH, fld, heap) \leftarrow$
 $\text{STORE}(base, fld, from), \text{VARPOINTS_TO}(from, heap),$
 $\text{VARPOINTS_TO}(base, baseH).$

$\text{VARPOINTS_TO}(to, heap) \leftarrow$
 $\text{LOAD}(to, base, fld), \text{VARPOINTS_TO}(base, baseH),$
 $\text{FLDPOINTS_TO}(baseH, fld, heap).$

$\text{INTERPROC_ASSIGN}(to, from) \leftarrow \text{CALL_GRAPH}(invo, meth),$
 $\text{FORMAL_ARG}(meth, n, to), \text{ACTUAL_ARG}(invo, n, from).$

$\text{INTERPROC_ASSIGN}(to, from) \leftarrow \text{CALL_GRAPH}(invo, meth),$
 $\text{FORMAL_RETURN}(meth, from), \text{ACTUAL_RETURN}(invo, to).$

$\text{VARPOINTS_TO}(to, heap) \leftarrow$
 $\text{INTERPROC_ASSIGN}(to, from), \text{VARPOINTS_TO}(from, heap).$

Example

```
Object f(Object p) {  
    return p;  
}  
a = new A();  
b = f(a);
```

Interprocedural Analysis (Higher-Order)

- Input relations:

ALLOC (*var* : *V*, *heap* : *H*, *inMeth* : *M*)

MOVE (*to* : *V*, *from* : *V*)

LOAD(*to* : *V*, *base* : *V*, *fld* : *F*)

STORE(*base* : *V*, *fld* : *F*, *from* : *V*)

VCALL(*base* : *V*, *sig* : *S*, *invo* : *I*, *inMeth* : *M*)

FORMALARG(*meth* : *M*, *i* : \mathbb{N} , *arg* : *V*)

ACTUALARG(*invo* : *I*, *i* : \mathbb{N} , *arg* : *V*)

FORMALRETURN(*meth* : *M*, *ret* : *V*)

ACTUALRETURN(*invo* : *I*, *var* : *V*)

THISVAR(*meth* : *M*, *this* : *V*)

HEAPTYPE(*heap* : *H*, *type* : *T*)

LOOKUP(*type* : *T*, *sig* : *S*, *meth* : *M*)

- Output relations:

VARPOINTSTO (*var* : *V*, *heap* : *H*)

FLDPOINTSTO (*baseH* : *H*, *fld* : *F*, *heap* : *H*)

INTERPROCASSIGN (*to* : *V*, *from* : *V*)

CALLGRAPH (*invo* : *I*, *meth* : *M*)

REACHABLE (*meth* : *M*)

Analysis Rules

A new rule for `VCALL` whose main job is to update call-graph:

```
REACHABLE(toMeth),  
VARPOINTSTO(this, heap),  
CALLGRAPH(invo, toMeth) ←  
  VCALL(base, sig, invo, inMeth), REACHABLE(inMeth),  
  VARPOINTSTO(base, heap),  
  HEAPTYPE(heap, heapT), LOOKUP(heapT, sig, toMeth),  
  THISVAR(toMeth, this).
```

The analysis performs on-the-fly call-graph construction.

- Pointer analysis and call-graph construction are closed inter-connected in object-oriented and higher-order languages. For example, to resolve call `obj.fun()`, we need pointer analysis. To compute points-to set of `a` in `f(Object a){...}`, we need call-graph.
- This global fixed point computation increases precision of both.

Example

```
class C {
    Object id(Object v){ return v; }
}
class B {
    void m (){
        C c = new C();
        D d = c.id(new D());
        E e = c.id(new E());
    }
}
public class A {
    void f(){
        B b = new B();
        b.m();
        b.m();
    }
}
```

Need for Context-Sensitivity

- Our current analysis is context-insensitive:

```
class C { Object id(Object v){ return v; } }  
class B {  
    void m (){  
        C c = new C();  
        D d = c.id(new D());  
        E e = c.id(new E()); } }  
public class A {  
    void f(){  
        B b = new B();  
        b.m();  
        b.m(); } }
```

- To achieve more precision, we can qualify the analysis results with context information. Two kinds of contexts:
 - ▶ *Calling context* for qualifying local variables
 - ▶ *Heap context* for qualifying heap abstractions

Domains for Context-Sensitive Analysis

- V is a set of program variables
- H is a set of heap abstractions (i.e. allocation sites)
- M is a set of method identifiers
- S is a set of method signatures (including name, type signature)
- F is a set of fields
- I is a set of instructions
- T is a set of class types
- \mathbb{N} is the set of natural numbers
- C is a set of calling contexts
- HC is a set of heap contexts

Output Relations for Context-Sensitive Analysis

The output relations are modified to add contexts:

$\text{VARPOINTS_TO}(var : V, ctx : C, heap : H, hctx : HC)$

$\text{CALLGRAPH}(invo : I, callerCtx : C, meth : M, calleeCtx : C)$

$\text{FLDPOINTS_TO}(baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC)$

$\text{INTERPROC_ASSIGN}(to : V, toCtx : C, from : V, fromCtx : C)$

$\text{REACHABLE}(meth : M, ctx : C)$

Context constructors:

Record $(heap : H, ctx : C) = newHCtx : HC$

Merge $(heap : H, hctx : HC, invo : I, ctx : C) = newCtx : C$

- **Record** generates heap contexts.
- **Merge** generates calling contexts.
- Different choices of them yield different context-sensitivity flavors.

Analysis Rules

Record (*heap*, *ctx*) = *hctx*,

VARPOINTSTO(*var*, *ctx*, *heap*, *hctx*) \leftarrow

REACHABLE(*meth*, *ctx*), ALLOC(*var*, *heap*, *meth*).

VARPOINTSTO(*to*, *ctx*, *heap*, *hctx*) \leftarrow

MOVE(*to*, *from*), VARPOINTSTO(*from*, *ctx*, *heap*, *hctx*).

FLDPOINTSTO(*baseH*, *baseHCtx*, *fld*, *heap*, *hctx*) \leftarrow

STORE(*base*, *fld*, *from*), VARPOINTSTO(*from*, *ctx*, *heap*, *hctx*),

VARPOINTSTO(*base*, *ctx*, *baseH*, *baseHCtx*).

VARPOINTSTO(*to*, *ctx*, *heap*, *hctx*) \leftarrow

LOAD(*to*, *base*, *fld*), VARPOINTSTO(*base*, *ctx*, *baseH*, *baseHCtx*),

FLDPOINTSTO(*baseH*, *baseHCtx*, *fld*, *heap*, *hctx*).

Analysis Rules

Merge (*heap, hctx, invo, callerCtx*) = *calleeCtx*,
REACHABLE(*toMeth, calleeCtx*),
VARPOINTSTO(*this, calleeCtx, heap, hctx*),
CALLGRAPH(*invo, callerCtx, toMeth, calleeCtx*) ←
VCALL(*base, sig, invo, inMeth*), REACHABLE(*inMeth, callerCtx*),
VARPOINTSTO(*base, callerCtx, heap, hctx*),
HEAPTYPE(*heap, heapT*), LOOKUP(*heapT, sig, toMeth*),
THISVAR(*toMeth, this*).

INTERPROCASSIGN(*to, calleeCtx, from, callerCtx*) ←
CALLGRAPH(*invo, callerCtx, meth, calleeCtx*),
FORMALARG(*meth, n, to*), ACTUALARG(*invo, n, from*).

INTERPROCASSIGN(*to, callerCtx, from, calleeCtx*) ←
CALLGRAPH(*invo, callerCtx, meth, calleeCtx*),
FORMALRETURN(*meth, from*), ACTUALRETURN(*invo, to*).

VARPOINTSTO(*to, toCtx, heap, hctx*) ←
INTERPROCASSIGN(*to, toCtx, from, fromCtx*),
VARPOINTSTO(*from, fromCtx, heap, hctx*).

Call-Site-Sensitivity (aka., k -CFA)

- The best-known flavor of context-sensitivity. It uses call-sites as contexts.
- In k -CFA, a method gets analyzed with the context that is a sequence of the last k call-sites (the current call-site of the method, the call-site of the caller method, the call-site of the caller method's caller, etc, up to a pre-defined depth, k).

Call-Site-Sensitivity

- 1-call-site sensitive with context-insensitive heap:

$$\begin{aligned}C &= I, & HC &= \{\star\} \\ \mathbf{Record}(heap, ctx) &= \star \\ \mathbf{Merge}(heap, hctx, invo, ctx) &= invo\end{aligned}$$

- 1-call-site sensitive with context-sensitive heap:

$$\begin{aligned}C &= I, & HC &= I \\ \mathbf{Record}(heap, ctx) &= ctx \\ \mathbf{Merge}(heap, hctx, invo, ctx) &= invo\end{aligned}$$

- 2-call-site sensitive with 1-call-site-sensitive heap:

$$\begin{aligned}C &= I \times I, & HC &= I \\ \mathbf{Record}(heap, ctx) &= \mathit{first}(ctx) \\ \mathbf{Merge}(heap, hctx, invo, ctx) &= \mathit{pair}(invo, \mathit{first}(ctx))\end{aligned}$$

Example

```
class C { Object id(Object v){ return v; } }
class B {
    void m (){
        C c = new C();
        D d = c.id(new D());
        E e = c.id(new E());
    }
}
public class A {
    void f(){
        B b = new B();
        b.m();
        b.m();
    }
}
```

Object-Sensitivity

- The dominant flavor of context-sensitivity for object-oriented languages.
- It uses object abstractions (i.e. allocation sites) as contexts, qualifying a method's local variables with the allocation site of the receiver object of the method call.

```
class A { void m() { return; } }
```

```
...
```

```
b = new B();
```

```
b.m();
```

The context of `m` is the allocation site of `b`.

Exercise

```
class S {
    Object id(Object a) { return a; }
    Object id2(Object a) { return id(); }
}

class C extends S {
    void fun1() {
        Object a1 = new A1();
        Object b1 = id2(a1);
    }
}

class D extends S {
    void fun2() {
        Object a2 = new A2();
        Object b2 = id2(a2);
    }
}
```

- What is the result of 1-call-site-sensitive analysis?
- What is the result of 1-object-sensitive analysis?
- Explain the strength of object-sensitivity over call-site-sensitivity.

Object-Sensitivity

- 1-object-sensitive with context-insensitive heap:

$$\begin{aligned}C &= H, & HC &= \{\star\} \\ \mathbf{Record}(heap, ctx) &= \star \\ \mathbf{Merge}(heap, hctx, invo, ctx) &= heap\end{aligned}$$

- 2-object-sensitive with 1-context-sensitive heap:

$$\begin{aligned}C &= H \times H, & HC &= H \\ \mathbf{Record}(heap, ctx) &= \mathit{first}(ctx) \\ \mathbf{Merge}(heap, hctx, invo, ctx) &= \mathit{pair}(heap, hctx)\end{aligned}$$

Example

```
class D{} class E{}
class C { Object id(Object v) { return v; } }
class B {
    Object id(Object v) {
        C c = new C(); // l3, heap objects: ([l1],l3), ([l2],l3)
        return c.id(v); // calling contexts: [l3,l1], [l3,l2]
    }
}
class A {
    void m () {
        B b1 = new B(); // l1
        B b2 = new B(); // l2
        D d = b1.id (new D()); // calling contexts: l1
        E e = b2.id (new E()); // calling contexts: l2
    }
}
```

Summary

We have covered a number of key concepts in program analysis:

- Pointer analysis
- Constraint-based analysis
- Interprocedural analysis
- Analysis of higher-order programs
- Context-sensitivity

For more details, see

- Yannis Smaragdakis and George Balatsouras. Pointer Analysis. Foundations and Trends in Programming Languages. 2(1). 2015.