IC637 Program Analysis

Lecture 5: Pointer Analysis

Minseok Jeon

2025 Fall

Example: Null Pointer Dereference Detection

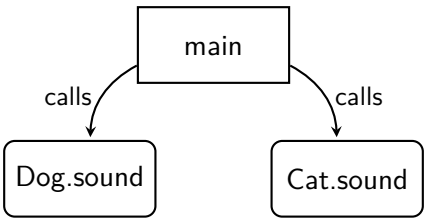
```
public class Example {
    static class Node {
        int value;
        Node next;
    }

    public static void main(String[] args) {
        Node head = null;
        head.next = new Node();
    }
}
```

Example: Call-Graph

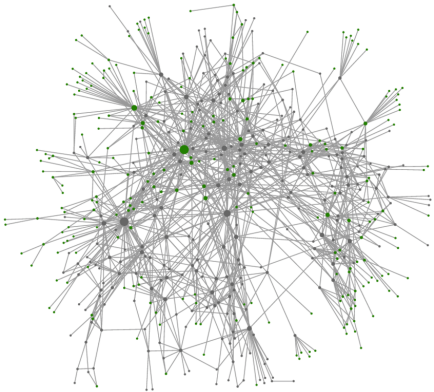
```
class Dog {
    void sound() {
        print("Woof!");
    }
}
class Cat {
    void sound() {
        print("Meow!");
    }
}
class Complicated {
    void sound() {
        foo1();
        ...
        foo999999();
    }
}
```

```
void main(String[] args) {
    Dog x = new Dog();
    Cat y = new Cat();
    x.sound();
    y.sound();
}
```

Example: Call-Graph



vs



Pointer Analysis

- Pointer analysis 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 about program properties need pointer analysis.
  - E.g., control-flows, data-flows, types, numeric values, etc
- Need for Pointer Analysis
- **Question:** how can we define an abstract domain for pointer analysis?

Abstraction of Memory Objects

- Memory locations are unbounded:

```
Object f(){
  x = new A();
  return x;
}

Object g(){
  y = new B();
  return y;
}

void main() {
  Object p = f();
  Object q = g();
}
```

- How can we define an abstract domain for pointer analysis?

Abstraction of Memory Objects

- In a typical pointer analysis, objects are abstracted into their allocation-sites.

```
Object f(){
  x = new A(); //l1
  return x;
}

Object g(){
  y = new B(); //l2
  return y;
}

void main() {
  Object p = f();
  Object q = g();
}
```

$$x \to \{l_1\} \quad y \to \{l_2\} \quad p \to \{l_1\} \quad q \to \{l_2\}$$

cf) Flow Sensitivity

- A flow-sensitive analysis maintains abstract states separately for each program point: e.g.,

$$1 : x = A()//l_1 \qquad 1 : x \to \{l_1\}$$
$$2 : x = B()//l_2 \qquad 2 : x \to \{l_2\}$$

- Pointer analysis is often defined flow-insensitively

$$1 : x = A()//l_1$$
$$2 : x = B()//l_2 \qquad x \to \{l_1, l_2\}$$

Constraint-based Analysis

- How can we express pointer analysis?

$$x = A()//l_1$$
$$y = x$$

Constraint-based Analysis

- Pointer analysis is expressed as subset constraints. The analysis is to compute the smallest solution of the constraints.

$$\begin{array}{lcl} x = A()//l_1 & & \{l_1\} \subseteq pts(x) \\ y = x & \Rightarrow & pts(x) \subseteq pts(y) \end{array}$$

- We use the Datalog language to express such constraints

Input and Output Relations

- A program is represented by a set of "facts" (relations):

$$Alloc(var : V, heap : H, inMeth : M) \qquad V : \text{the set of program variables}$$
$$Move(to : V, from : V, inMeth : M) \qquad H : \text{the set of heap locations}$$
$$Load(to : V, base : V, fld : F, inMeth : M) \qquad F : \text{the set of fields}$$
$$Store(base : V, fld : F, from : V, inMeth : M) \qquad M : \text{the set of method identifiers}$$

- Example:

```
main(){ //m1
  a = A(); // l1
  b = B(); // l2
  c = a;
  a.f = b;
  d = c.f;
}
```

$\Rightarrow$

$Alloc(a, l_1, m_1)$ (from a = A();//l_1)
$Alloc(b, l_2, m_1)$ (from b = B();//l_2)
$Move(c, a, m_1)$ (from c = a)
$Store(a, f, b, m_1)$ (from a.f = b)
$Load(d, c, f, m_1)$ (from d = c.f)

Input and Output Relations

- Output relations:

$$VarPointsTo(var : V, heap : H)$$
$$FldPointsTo(baseH : H, fld : F, heap : H)$$

- Example:

```
main(){ //m1
  a = A(); // l1
  b = B(); // l2
  c = a;
  a.f = b;
  d = c.f;
}
```

$\Rightarrow$

$$Alloc(a, l_1, m_1)$$
$$Alloc(b, l_2, m_1)$$
$$Move(c, a, m_1)$$
$$Store(a, f, b, m_1)$$
$$Load(d, c, f, m_1)$$

$\Rightarrow$

$$VarPointsTo(a, l_1)$$
$$VarPointsTo(b, l_2)$$
$$VarPointsTo(c, l_1)$$
$$FldPointsTo(l_1, f, l_2)$$
$$VarPointsTo(d, l_2)$$

Quiz

- Does pointer analysis always terminate? If so or not, why?

Fixed Point Computation

(1) $VarPointsTo(var,\ heap) \leftarrow Alloc(var,\ heap,\ inMeth)$

(2) $VarPointsTo(to,\ heap) \leftarrow$
$Move(to,\ from,\ inMeth),\ VarPointsTo(from,\ heap)$

(3) $FldPointsTo(baseH,\ fld,\ heap) \leftarrow$
$Store(base,\ fld,\ from,\ inMeth),\ VarPointsTo(from,\ heap),$
$VarPointsTo(base,\ baseH)$

(4) $VarPointsTo(to,\ heap) \leftarrow$
$Load(to,\ base,\ fld,\ inMeth),\ VarPointsTo(base,\ baseH),$
$FldPointsTo(baseH,\ fld,\ heap)$

Fixed Point Computation

$$
\begin{array}{r}
Alloc(a, l_1, m_1) \\
Alloc(b, l_2, m_1) \\
Move(c, a, m_1) \\
Store(a, f, b, m_1) \\
Load(d, c, f, m_1)
\end{array}
\quad \begin{array}{c} (1) \\ \Rightarrow \end{array} \quad
\begin{array}{r}
Alloc(a, l_1, m_1) \\
Alloc(b, l_2, m_1) \\
Move(c, a, m_1) \\
Store(a, f, b, m_1) \\
Load(d, c, f, m_1) \\
VarPointsTo(a, l_1) \\
VarPointsTo(b, l_2)
\end{array}
$$

Fixed Point Computation

(2), (3) $\Rightarrow$

$$
\begin{aligned}
Alloc(a, l_1, m_1) \\
Alloc(b, l_2, m_1) \\
Move(c, a, m_1) \\
Store(a, f, b, m_1) \\
Load(d, c, f, m_1) \\
VarPointsTo(a, l_1) \\
VarPointsTo(b, l_2) \\
VarPointsTo(c, l_1) \\
FldPointsTo(l_1, f, l_2)
\end{aligned}
$$

(4) $\Rightarrow$

$$
\begin{aligned}
Alloc(a, l_1, m_1) \\
Alloc(b, l_2, m_1) \\
Move(c, a, m_1) \\
Store(a, f, b, m_1) \\
Load(d, c, f, m_1) \\
VarPointsTo(a, l_1) \\
VarPointsTo(b, l_2) \\
VarPointsTo(c, l_1) \\
FldPointsTo(l_1, f, l_2) \\
VarPointsTo(d, l_2)
\end{aligned}
$$

Interprocedural Analysis

- Example code and Input relations.

```
class C{
  Object id(v){ //m2
    this = this;
    return v;
  }
}
class D{
  void main(){ //m1
    c = new C(); //l1
    a = new A(); //l2
    b = c.id(a); //l3
  }
}
```

$\Rightarrow$

$FormalParam(m_2, 0, v)$	(from id(v))
$FormalReturn(m_2, v)$	(from id(v) ... return v)
$Alloc(c, l_1)$	(from c = new C();//l_1)
$Alloc(a, l_2)$	(from a = new A();//l_2)
$ActualParam(l_3, 0, a)$	(from b = id(a)//l_3)
$ActualReturn(l_3, b)$	(from b = id(a)//l_3)
$VirtualCall(c, id, l_3, m_1)$	(from b = c.id(a);//l_3)
$HeapType(l_1, C)$	(from c = new C();//l_1)
$HeapType(l_2, A)$	(from a = new A();//l_2)
$ThisVar(m_2, this)$	(from class C...id(v))
$Lookup(C, id, m_2)$	(from class C...id(v))
$CallGraph(\_, m_1)$	(from main)

Interprocedural Analysis

- Output relations.

```
class C{
  Object id(v){ //m2
    this = this;
    return v;
  }
}
class D{
  void main(){ //m1
    c = new C(); //l1
    a = new A(); //l2
    b = c.id(a); //l3
  }
}
```

$\Rightarrow$

$$
\begin{aligned}
FormalParam(m_2, 0, v)\\
FormalReturn(m_2, v)\\
Alloc(c, l_1)\\
Alloc(a, l_2)\\
ActualParam(l_3, 0, a)\\
ActualReturn(l_3, b)\\
VirtualCall(c, id, l_3, m_1)\\
HeapType(l_1, C)\\
HeapType(l_2, A)\\
ThisVar(m_2, this)\\
Lookup(C, id, m_2)\\
CallGraph(\_, m_1)
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
CallGraph(\_, m_1)\\
CallGraph(l_2, m_2)\\
VarPointsTo(c, l_1)\\
VarPointsTo(this, l_1)\\
VarPointsTo(a, l_2)\\
VarPointsTo(v, l_2)\\
VarPointsTo(b, l_2)
\end{aligned}
$$

Input Relations

- Input relations program (representation):

$$Alloc(var : V, heap : H, inMeth : M)$$
$$Move(to : V, from : V, inMeth : M)$$
$$Load(to : V, base : V, fld : F, inMeth : M)$$
$$Store(base : V, fld : F, from : V, inMeth : M)$$
$$FormalParam(method : M, arg : A, var : V)$$
$$FormalReturn(method : M, var : V)$$
$$ActualParam(invocation : I, arg : A, var : V)$$
$$ActualReturn(invocation : I, var : V)$$
$$ThisVar(meth : M, this : V)$$
$$HeapType(heap : H, type : T)$$
$$LookUp(type : T, sig : S, methM)$$
$$VirtualCall(base : V, sig : S, invo : I, inMeth : M)$$

V : the set of program variables
H : the set of heap locations
F : the set of fields
M : the set of method identifiers
S : the set of method signatures (names)
I : the set of instructions
T : the set of class types
$\mathbb{N}$: the set of natural numbers

Output Relations

- Output relations (analysis results):

$$VarPointsTo(var : V, heap : H)$$
$$FldPointsTo(baseH : H, fld : F, heap : H)$$
$$CallGraph(invo : I, meth : M)$$

Analysis Rules

(1) $VarPointsTo(var,\ heap) \leftarrow$

$$Alloc(var,\ heap,\ inMeth), CallGraph(\_,\ inMeth)$$

(2) $VarPointsTo(to,\ heap) \leftarrow$

$$Move(to,\ from,\ inMeth),\ VarPointsTo(from,\ heap), CallGraph(\_,\ inMeth)$$

(3) $FldPointsTo(baseH,\ fld,\ heap) \leftarrow$

$$Store(base,\ fld,\ from,\ inMeth),\ VarPointsTo(from,\ heap),$$
$$VarPointsTo(base,\ baseH),\ CallGraph(\_,\ inMeth)$$

(4) $VarPointsTo(to,\ heap) \leftarrow$

$$Load(to,\ base,\ fld,\ inMeth),\ VarPointsTo(base,\ baseH),$$
$$FldPointsTo(baseH,\ fld,\ heap),\ CallGraph(\_,\ inMeth)$$

Analysis Rules

$$(5)\ VarPointsTo(this,\ heap),\ CallGraph(invo,\ toMeth) \leftarrow$$
$$VirtualCall(base,\ sig,\ invo,\ inMeth), VarPointsTo(base,\ BaseH),$$
$$CallGraph(\_,\ inMeth), HeapType(BaseH,\ type),$$
$$LookUp(type,\ sig,\ toMeth), ThisVar(toMeth,\ this)$$

$$(6)\ VarPointsTo(param',\ heap) \leftarrow$$
$$CallGraph(invo,\ meth), FormalParam(meth,\ n,\ param'),$$
$$ActualParam(invo,\ n,\ param), VarPointsTo(param,\ heap)$$

$$(7)\ VarPointsTo(return,\ heap) \leftarrow$$
$$CallGraph(invo,\ meth), FormalReturn(meth,\ return'),$$
$$ActualReturn(invo,\ return), VarPointsTo(return',\ heap)$$

Example

- Compute fixpoint of the analysis rules.

```
class C{
  Object id(v){ //m2
    this = this;
    return v;
  }
}
class D{
  void main(){ //m1
    c = new C(); //l1
    a = new A(); //l2
    b = c.id(a); //l3
  }
}
```

Example

- Compute fixpoint of the analysis rules.

$$FormalParam(m_2, 0, v)$$
$$FormalReturn(m_2, v)$$
$$Alloc(c, l_1, m_1)$$
$$Alloc(a, l_2, m_1)$$
$$ActualParam(l_3, 0, a)$$
$$ActualReturn(l_3, b)$$
$$VirtualCall(c, id, l_3, m_1)$$
$$HeapType(l_1, C)$$
$$HeapType(l_2, A)$$
$$ThisVar(m_2, this)$$
$$Lookup(C, id, m_2)$$
$$CallGraph(\_, m_1)$$

Static Call and Special Call

- Example code and Input relations.

```
class C{
  Object fld;
  C(Object value) {//Special Method
    this.fld = value;
  }
  static Object id(Object o) {//Static Method
    return o;
  }
}
class D{
  void main(){ //m1
    a = new A(); //l1
    c = new C(a); //l2
    b = C.id(c.fld); //l3
  }
}
```

Input Relations

- Input relations program (representation):

$$Alloc(var : V, heap : H, inMeth : M)$$
$$Move(to : V, from : V, inMeth : M)$$
$$Load(to : V, base : V, fld : F, inMeth : M)$$
$$Store(base : V, fld : F, from : V, inMeth : M)$$
$$FormalParam(method : M, arg : A, var : V)$$
$$FormalReturn(method : M, var : V)$$
$$ActualParam(invocation : I, arg : A, var : V)$$
$$ActualReturn(invocation : I, var : V)$$
$$\dots$$
$$VirtualCall(base : V, sig : S, invo : I, inMeth : M)$$
$$SpecialCall(base : V, invo : I, toMeth : M, inMeth : M)$$
$$StaticCall(invo : I, toMeth : M, inMeth : M)$$

V : the set of program variables
H : the set of heap locations
F : the set of fields
M : the set of method identifiers
S : the set of method signatures (names)
I : the set of instructions
T : the set of class types
$\mathbb{N}$: the set of natural numbers

Analysis Rules

$$(8) CallGraph(invo,\ toMeth) \leftarrow StaticCall(invo,\ toMeth,\ inMeth), CallGraph(\_,\ inMeth)$$

$$(9)\ VarPointsTo(this,\ heap),\ CallGraph(invo,\ toMeth) \leftarrow SpecialCall(base,\ invo,\ toMeth,\ inMeth), CallGraph(\_,\ inMeth), ThisVar(toMeth,\ this), VarPointsTo(base,\ heap)$$

Example 1 (Alloc)

- Compute the analysis result.

```
class A {
    int value;
    A(int value) {
        this.value = value;
    }
}
class B {
    String data;
    B(String data) {
        this.data = data;
    }
}
public class Alloc {
    public static void main(String[] args) {
        A objA1 = new A(10);
        A objA2 = new A(20);

        B objB1 = new B("Hello");
        B objB2 = new B("World");

    }
}
```

Example 2 (Move)

- Compute the analysis result.

```
class A {}

class B {}

public class Move {
    public static void main(String[] args) {
        A objA = new A();
        B objB = new B();

        Object v1;
        Object v2;

        if (args.length > 0) {
            v1 = objB;
            v2 = objA;
        } else {
            v1 = objA;
            v2 = objB;
        }
    }
}
```

Example 3 (Load)

- Compute the analysis result.

```
class A {
    Object fld;
}

class B {
    Object fld;
}

public class Load {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.fld = b;
        b.fld = a;

        Object o1 = a.fld;
        Object o2 = b.fld;
        assert o1 != o2 : "o1 should not be the same object as o2";
    }
}
```

Example 4 (Store)

- Compute the analysis result.

```
class A {
    Object fld;
}

class B {
    Object fld;
}

public class Store {
    public static void main(String[] args) {
        A a = new A();
        a.fld = new B();
        ((B) (a.fld)).fld = new A();
    }
}
```

Example 5 (Static Call)

- Compute the analysis result.

```
class A {}

class B {}

public class StaticCall {
    static Object id(Object o) { return o; }
    public static void main(String[] args) {
        A objA1 = new A();
        B objB1 = new B();
        A v1 = (A) id(objA1);
        B v2 = (B) id(objB1);
    }
}
```

Example 6 (Special Call)

- Compute the analysis result.

```
class A {
    Object fld;
    A(Object value) {
        this.fld = value;
    }
}

class B {
    Object fld;
    B(Object value) {
        this.fld = value;
    }
}

class C{}

public class SpecialCall {
    static Object id(Object o) { return o; }
    public static void main(String[] args) {
        C objC = new C();
        A a = new A(objC);
        B b = new B(objC);
    }
}
```

Example 7 (Virtual Call)

- Compute the analysis result.

```
class A {}

class B {}

class C {
    Object id(Object v){
        return v;
    }
}

public class VirtualCall {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();

        Object v1 = c.id(a);
        Object v2 = c.id(b);

        assert v1 != v2 : "v1 should not be the same object as v2";
    }
}
```

Lecture Wrap-Up

- **Pointer Analysis Fundamentals**
  - Computes the set of memory locations that pointer variables may point to
  - Essential for control-flow, data-flow, and other program analyses
- **Key Concepts**
  - Allocation-site abstraction for heap objects
  - Constraint-based formulation
- **Analysis Components**
  - Input relations: Alloc, Move, Load, Store, VirtualCall, etc.
  - Output relations: VarPointsTo, FldPointsTo, CallGraph
  - Fixed-point computation over inference rules
- **Interprocedural Analysis**
  - Handling method calls: virtual, static, and special calls
  - Parameter passing and return value propagation