Pointer Analysis

Hakjoo Oh
Korea University

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

Abstraction of Memory Objects

- Memory locations are unbounded:

```
def id (p): return p

def f():
  x = A()     // l1
  y = id(x)

def g():
  a = B()     // l2
  b = id(a)

while True: {f(); g()}
```

Abstraction of Memory Objects

- Memory locations are unbounded:

```
def id (p): return p

def f():
  x = A()     // l1
  y = id(x)

def g():
  a = B()     // l2
  b = id(a)

while True: {f(); g()}
```

- In a typical pointer analysis, objects are abstracted into their **allocation-sites**. Pointer analysis result:

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

cf) Flow Sensitivity

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

```
x = A()
y = id(x)
x = B()
y = id(x)
```

- Pointer analysis is often defined flow-insensitively

Constraint-based Analysis

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

```
x = A() // l1
y = x
```

$\Longrightarrow$

$$\{l_1\} \subseteq pts(x)$$
$$pts(x) \subseteq pts(y)$$

- We use the Datalog language to express such constraints

Input and Output Relations

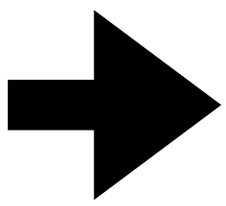
- A program is represented by a set of "facts" (relations):

Alloc($var : V, heap : H$)
Move($to : V, from : V$)
Load($to : V, base : V, fld : F$)
Store($base : V, fld : F, from : V$)

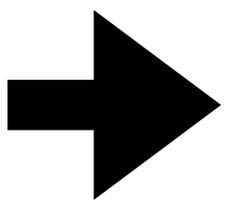
V: the set of program variables
H: the set of allocation sites
F: the set of field names

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

```
a = A() // l1
b = B() // l2
c = a
a.f = b
d = c.f
```

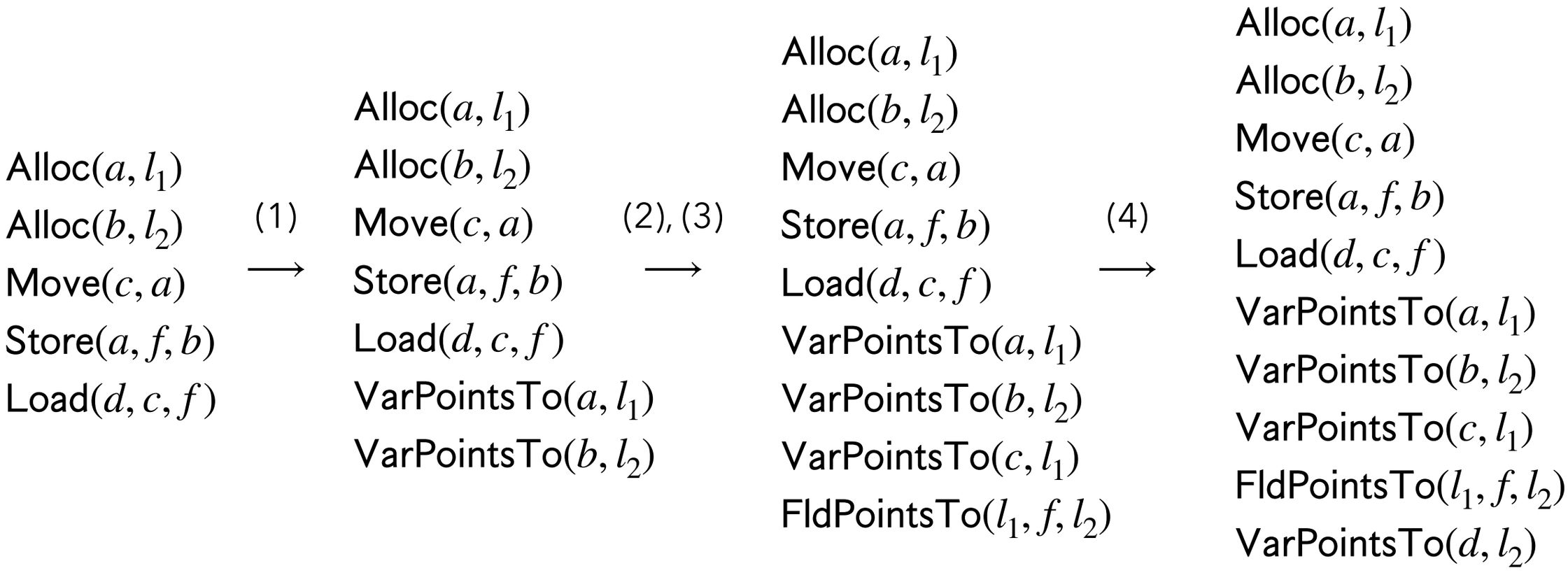


Alloc(a, l_1)
Alloc(b, l_2)
Move(c, a)
Store(a, f, b)
Load(d, c, f)



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

Fixed Point Computation



Pointer Analysis Rules

(1) VarPointsTo(var, $heap$) ← Alloc(var, $heap$)

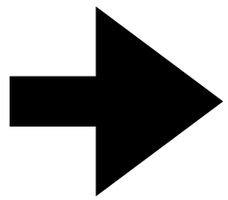
(2) VarPointsTo(to, $heap$) ←
Move(to, $from$), VarPointsTo($from$, $heap$)

(3) FldPointsTo($baseH$, fld, $heap$) ←
Store($base$, fld, $from$), VarPointsTo($from$, $heap$),
VarPointsTo($base$, $baseH$)

(4) VarPointsTo(to, $heap$) ←
Load(to, $base$, fld), VarPointsTo($base$, $baseH$),
FldPointsTo($baseH$, fld, $heap$)

Interprocedural Analysis (First-Order)

```
def f(p):   // m1
  return p
a = A()     // l1
b = f(a)    // l2
```



FormalArg(m_1,0,p)

FormalReturn(m_1, p)

Alloc(a, l_1, $global$)

CallGraph(l_2, m_1)

Reachable($global$)

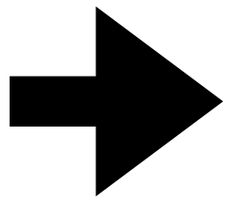
Reachable(m_1)

ActualArg(l_2,0,a)

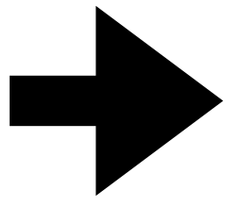
ActualReturn(l_2, b)

Interprocedural Analysis (First-Order)

```
def f(p):   // m1
  return p
a = A()     // l1
b = f(a)    // l2
```



FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc(a, l_1, $global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)



InterProcAssign(p, a)
InterProcAssign(b, p)
VarPointsTo(a, l_1)
VarPointsTo(p, l_1)
VarPointsTo(b, l_1)

Input and Output Relations

- Input relations (program representation)

  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($invo : I, var : V$)

V: the set of program variables

H: the set of allocation sites

F: the set of field names

M: the set of method identifiers

S: the set of method signatures

I: the set of instructions

T: the set of class types

$\mathbb{N}$: the set of natural numbers

- Output relations

  VarPointsTo($var : V, heap : H$)

  FldPointsTo($baseH : H, fld : F, heap : H$)

  InterProcAssign($to : V, from : V$)

Fixed Point Computation

FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc(a, l_1, $global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)

(1), (5), (6) ⟶

FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc(a, l_1, $global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)
VarPointsTo(a, l_1)
InterProcAssign(p, a)
InterProcAssign(b, p)

(7) ⟶*

FormalArg(m_1,0,p)
FormalReturn(m_1, p)
Alloc(a, l_1, $global$)
CallGraph(l_2, m_1)
Reachable($global$)
Reachable(m_1)
ActualArg(l_2,0,a)
ActualReturn(l_2, b)
VarPointsTo(a, l_1)
InterProcAssign(p, a)
InterProcAssign(b, p)
VarPointsTo(p, l_1)
VarPointsTo(b, l_1)

Analysis Rules

(1) VarPointsTo($var, heap$) ← Reachable($meth$), Alloc($var, heap, meth$)

(2) VarPointsTo($to, heap$) ←
Move($to, from$), VarPointsTo($from, heap$)

(3) FldPointsTo($baseH, fld, heap$) ←
Store($base, fld, from$), VarPointsTo($from, heap$), VarPointsTo($base, baseH$)

(4) VarPointsTo($to, heap$) ←
Load($to, base, fld$), VarPointsTo($base, baseH$), FldPointsTo($baseH, fld, heap$)

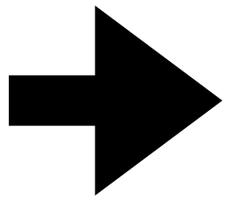
(5) InterProcAssign($to, from$) ←
CallGraph($invo, meth$), FormalArg($meth, n, to$), ActualArg($invo, n, from$)

(6) InterProcAssign($to, from$) ←
CallGraph($invo, meth$), FormalReturn($meth, from$), ActualReturn($invo, to$)

(7) VarPointsTo($to, heap$) ←
InterProcAssign($to, from$), VarPointsTo($from, heap$)

Interprocedural Analysis (Higher-Order)

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

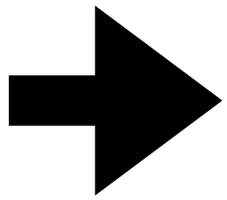


FormalArg(m_1,0,v)
FormalReturn(m_1, v)
ThisVar(m_1, $self$)
LookUp(C, id, m_1)
ThisVar(m_2, $self$)
LookUp(B, g, m_2)
Alloc(c, l_1, m_2)
Alloc(s, l_2, m_2)
Alloc(t, l_3, m_2)
HeapType(l_1, C)
HeapType(l_2, D)
HeapType(l_3, E)

VCall(c, id, l_4, m_2)
VCall(c, id, l_5, m_2)
ActualArg(l_4,0,s)
ActualArg(l_5,0,t)
ActualReturn(l_4, d)
ActualReturn(l_5, e)
ThisVar(m_3, $self$)
LookUp(A, f, m_3)
Alloc(b, l_6, m_3)
HeapType(l_6, B)
VCall(b, g, l_7, m_3)
VCall(b, g, l_8, m_3)
Reachable(m_3)

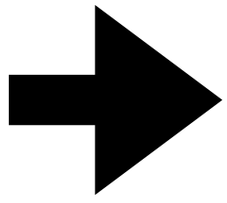
Interprocedural Analysis (Higher-Order)

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):          // m2
    c = C()             // l1
    s = D()             // l2
    t = E()             // l3
    d = c.id(s)         // l4
    e = c.id(t)         // l5

class A:
  def f(self):          // m3
    b = B()             // l6
    b.g()               // l7
    b.g()               // l8
```



FormalArg(m_1,0,v)
FormalReturn(m_1, v)
ThisVar(m_1, $self$)
LookUp(C, id, m_1)
ThisVar(m_2, $self$)
LookUp(B, g, m_2)
Alloc(c, l_1, m_2)
Alloc(s, l_2, m_2)
Alloc(t, l_3, m_2)
HeapType(l_1, C)
HeapType(l_2, D)
HeapType(l_3, E)

VarPointsTo(b, l_6)
Reachable(m_2)
VarPointsTo($self$, l_6)
CallGraph(l_7, m_2)
CallGraph(l_8, m_2)
VarPointsTo(c, l_1)
VarPointsTo(s, l_2)
VarPointsTo(t, l_3)
Reachable(m_1)
VarPointsTo($self$, l_1)
CallGraph(l_4, m_1)
CallGraph(l_5, m_1)



VCall(c, id, l_4, m_2)
VCall(c, id, l_5, m_2)
ActualArg(l_4,0,s)
ActualArg(l_5,0,t)
ActualReturn(l_4, d)
ActualReturn(l_5, e)
ThisVar(m_3, $self$)
LookUp(A, f, m_3)
Alloc(b, l_6, m_3)
HeapType(l_6, B)
VCall(b, g, l_7, m_3)
VCall(b, g, l_8, m_3)
Reachable(m_3)

InterProcAssign(v, s)
InterProcAssign(v, t)
InterProcAssign(d, v)
InterProcAssign(e, v)
VarPointsTo(v, l_2)
VarPointsTo(v, l_3)
VarPointsTo(d, l_2)
VarPointsTo(d, l_3)
VarPointsTo(e, l_2)
VarPointsTo(e, l_3)

Input and Output Relations

- 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

(1) VarPointsTo($var, heap$) ← Reachable($meth$), Alloc($var, heap, meth$)

(2) VarPointsTo($to, heap$) ←
Move($to, from$), VarPointsTo($from, heap$)

(3) FldPointsTo($baseH, fld, heap$) ←
Store($base, fld, from$), VarPointsTo($from, heap$), VarPointsTo($base, baseH$)

(4) VarPointsTo($to, heap$) ←
Load($to, base, fld$), VarPointsTo($base, baseH$), FldPointsTo($baseH, fld, heap$)

(5) InterProcAssign($to, from$) ←
CallGraph($invo, meth$), FormalArg($meth, n, to$), ActualArg($invo, n, from$)

(6) InterProcAssign($to, from$) ←
CallGraph($invo, meth$), FormalReturn($meth, from$), ActualReturn($invo, to$)

(7) VarPointsTo($to, heap$) ←
InterProcAssign($to, from$), VarPointsTo($from, heap$)

Analysis Rules

(8) 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($toMeth$, $this$)

Analysis Rules

(8) 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($toMeth$, $this$)

- This analysis performs **on-the-fly call-graph construction.** Pointer analysis and call-graph construction are closely 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.

FormalArg(m_1,0,v)
FormalReturn(m_1, v)
ThisVar(m_1, *self*)
LookUp(C, id, m_1)
ThisVar(m_2, *self*)
LookUp(B, g, m_2)
Alloc(c, l_1, m_2)
Alloc(s, l_2, m_2)
Alloc(t, l_3, m_2)
HeapType(l_1, C)
HeapType(l_2, D)
HeapType(l_3, E)
VCall(c, id, l_4, m_2)
VCall(c, id, l_5, m_2)
ActualArg(l_4,0,s)
ActualArg(l_5,0,t)
ActualReturn(l_4, d)
ActualReturn(l_5, e)
ThisVar(m_3, *self*)
LookUp(A, f, m_3)
Alloc(b, l_6, m_3)
HeapType(l_6, B)
VCall(b, g, l_7, m_3)
VCall(b, g, l_8, m_3)
Reachable(m_3)

(1) ⟶ VarPointsTo(b, l_6)

(8) ⟶ Reachable(m_1)
VarPointsTo(*self*, l_1)
CallGraph(l_4, m_1)
CallGraph(l_5, m_1)

(7) ⟶ VarPointsTo(d, l_2)
VarPointsTo(d, l_3)
VarPointsTo(e, l_2)
VarPointsTo(e, l_3)

(8) ⟶ Reachable(m_2)
VarPointsTo(*self*, l_6)
CallGraph(l_7, m_2)
CallGraph(l_8, m_2)

(5), (6) ⟶ InterProcAssign(v, s)
InterProcAssign(v, t)
InterProcAssign(d, v)
InterProcAssign(e, v)

(1) ⟶ VarPointsTo(c, l_1)
VarPointsTo(s, l_2)
VarPointsTo(t, l_3)

(7) ⟶ VarPointsTo(v, l_2)
VarPointsTo(v, l_3)

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

Context Sensitivity

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):          // m2
    c = C()             // l1
    s = D()             // l2
    t = E()             // l3
    d = c.id(s)         // l4
    e = c.id(t)         // l5

class A:
  def f(self):          // m3
    b = B()             // l6
    b.g()               // l7
    b.g()               // l8
```

VarPointsTo(b, l_6)
VarPointsTo($self, l_6$)
VarPointsTo(c, l_1)
VarPointsTo(s, l_2)
VarPointsTo(t, l_3)
VarPointsTo($self, l_1$)
VarPointsTo(v, l_2)
VarPointsTo(v, l_3)
VarPointsTo(d, l_2)
VarPointsTo(d, l_3)
VarPointsTo(e, l_2)
VarPointsTo(e, l_3)

context-insensitive

VarPointsTo($b, \star, l_6, \star$)
VarPointsTo($self, l_7, l_6, \star$)
VarPointsTo($self, l_8, l_6, \star$)
VarPointsTo(c, l_7, l_1, l_7)
VarPointsTo(s, l_7, l_2, l_7)
VarPointsTo(t, l_7, l_3, l_7)
VarPointsTo(c, l_8, l_1, l_8)
VarPointsTo(s, l_8, l_2, l_8)
VarPointsTo(t, l_8, l_3, l_8)
VarPointsTo($self, l_4 \cdot l_7, l_1, l_7$)
VarPointsTo($self, l_4 \cdot l_8, l_1, l_8$)
VarPointsTo($self, l_5 \cdot l_7, l_1, l_7$)
VarPointsTo($self, l_5 \cdot l_8, l_1, l_8$)
VarPointsTo($v, l_4 \cdot l_7, l_2, l_7$)
VarPointsTo($v, l_4 \cdot l_8, l_2, l_8$)
VarPointsTo($v, l_5 \cdot l_7, l_3, l_7$)
VarPointsTo($v, l_5 \cdot l_8, l_3, l_8$)
VarPointsTo(d, l_7, l_2, l_7)
VarPointsTo(d, l_8, l_2, l_8)
VarPointsTo(e, l_7, l_3, l_7)
VarPointsTo(e, l_8, l_3, l_8)

context-sensitive
(2-call-site sensitivity with
context-sensitive heap)

Context Sensitivity

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

1-call-site sensitivity
w/ context-sensitive heap

VarPointsTo(b, $\star$,l_6, $\star$)
VarPointsTo($self$, l_7, l_6, $\star$)
VarPointsTo($self$, l_8, l_6, $\star$)
VarPointsTo(c, l_7, l_1, l_7)
VarPointsTo(s, l_7, l_2, l_7)
VarPointsTo(t, l_7, l_3, l_7)
VarPointsTo(c, l_8, l_1, l_8)
VarPointsTo(s, l_8, l_2, l_8)
VarPointsTo(t, l_8, l_3, l_8)
VarPointsTo($self$, l_4, l_1, l_7)
VarPointsTo($self$, l_4, l_1, l_8)
VarPointsTo($self$, l_5, l_1, l_7)
VarPointsTo($self$, l_5, l_1, l_8)
VarPointsTo(v, l_4, l_2, l_7)
VarPointsTo(v, l_4, l_2, l_8)
VarPointsTo(v, l_5, l_3, l_7)
VarPointsTo(v, l_5, l_3, l_8)
VarPointsTo(d, l_7, l_2, l_7)
VarPointsTo(d, l_7, l_2, l_8)
VarPointsTo(d, l_8, l_2, l_7)
VarPointsTo(d, l_8, l_2, l_8)
VarPointsTo(e, l_7, l_3, l_7)
VarPointsTo(e, l_7, l_3, l_8)
VarPointsTo(e, l_8, l_3, l_7)
VarPointsTo(e, l_8, l_3, l_8)

Context Sensitivity

```
class C:
  def id(self, v): // m1
    return v

class B:
  def g(self):      // m2
    c = C()         // l1
    s = D()         // l2
    t = E()         // l3
    d = c.id(s)     // l4
    e = c.id(t)     // l5

class A:
  def f(self):      // m3
    b = B()         // l6
    b.g()           // l7
    b.g()           // l8
```

VarPointsTo(b, $\star$,l_6, $\star$)
VarPointsTo($self$, l_7, l_6, $\star$)
VarPointsTo($self$, l_8, l_6, $\star$)
VarPointsTo(c, l_7, l_1, $\star$)
VarPointsTo(s, l_7, l_2, $\star$)
VarPointsTo(t, l_7, l_3, $\star$)
VarPointsTo(c, l_8, l_1, $\star$)
VarPointsTo(s, l_8, l_2, $\star$)
VarPointsTo(t, l_8, l_3, $\star$)
VarPointsTo($self$, l_4, l_1, $\star$)
VarPointsTo($self$, l_5, l_1, $\star$)
VarPointsTo(v, l_4, l_2, $\star$)
VarPointsTo(v, l_5, l_3, $\star$)
VarPointsTo(d, l_7, l_2, $\star$)
VarPointsTo(d, l_8, l_2, $\star$)
VarPointsTo(e, l_7, l_3, $\star$)
VarPointsTo(e, l_8, l_3, $\star$)

1-call-site sensitivity
w/ context-insensitive heap

VarPointsTo(b, $\star$,l_6, $\star$)
VarPointsTo($self$, l_7, l_6, $\star$)
VarPointsTo($self$, l_8, l_6, $\star$)
VarPointsTo(c, l_7, l_1, $\star$)
VarPointsTo(s, l_7, l_2, $\star$)
VarPointsTo(t, l_7, l_3, $\star$)
VarPointsTo(c, l_8, l_1, $\star$)
VarPointsTo(s, l_8, l_2, $\star$)
VarPointsTo(t, l_8, l_3, $\star$)
VarPointsTo($self$, $l_4 \cdot l_7$, l_1, $\star$)
VarPointsTo($self$, $l_4 \cdot l_8$, l_1, $\star$)
VarPointsTo($self$, $l_5 \cdot l_7$, l_1, $\star$)
VarPointsTo($self$, $l_5 \cdot l_8$, l_1, $\star$)
VarPointsTo(v, $l_4 \cdot l_7$, l_2, $\star$)
VarPointsTo(v, $l_4 \cdot l_8$, l_2, $\star$)
VarPointsTo(v, $l_5 \cdot l_7$, l_3, $\star$)
VarPointsTo(v, $l_5 \cdot l_8$, l_3, $\star$)
VarPointsTo(d, l_7, l_2, $\star$)
VarPointsTo(d, l_8, l_2, $\star$)
VarPointsTo(e, l_7, l_3, $\star$)
VarPointsTo(e, l_8, l_3, $\star$)

2-call-site sensitivity
w/ context-insensitive heap

Object Sensitivity

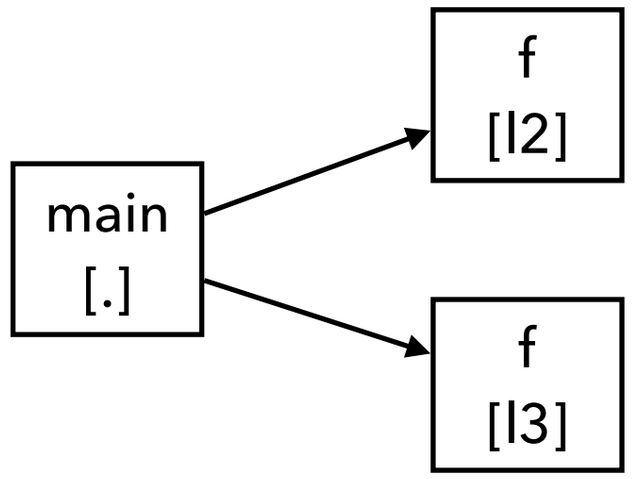
- 2-object sensitivity with 1-call-site sensitive heap:

```
class C:
  def h(self):
     return

class B:
  def g(self):
    c = C()        // l3, heap objects: (l3, [l1]), (l3, [l2])
    c.h()          // contexts: (l3, l1), (l3, l2)

class A:
  def f(self):
    b1 = B()       // l1
    b2 = B()       // l2
    b1.g()         // context: l1
    b2.g()         // context: l2
```

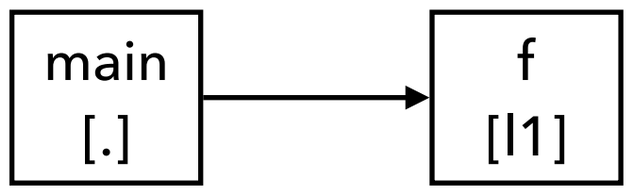
Call-site vs. Object Sensitivity

- Typical example that benefits from call-site sensitivity:

```
class A:
  def f(self): return

def main():
   a = A()  // l1
   a.f()    // l2
   a.f()    // l3
```



call-site sensitivity

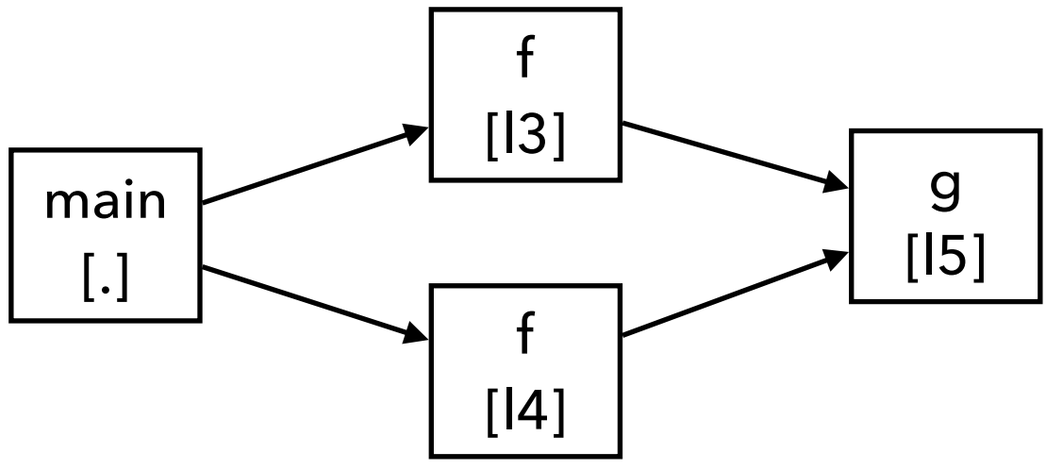


object sensitivity

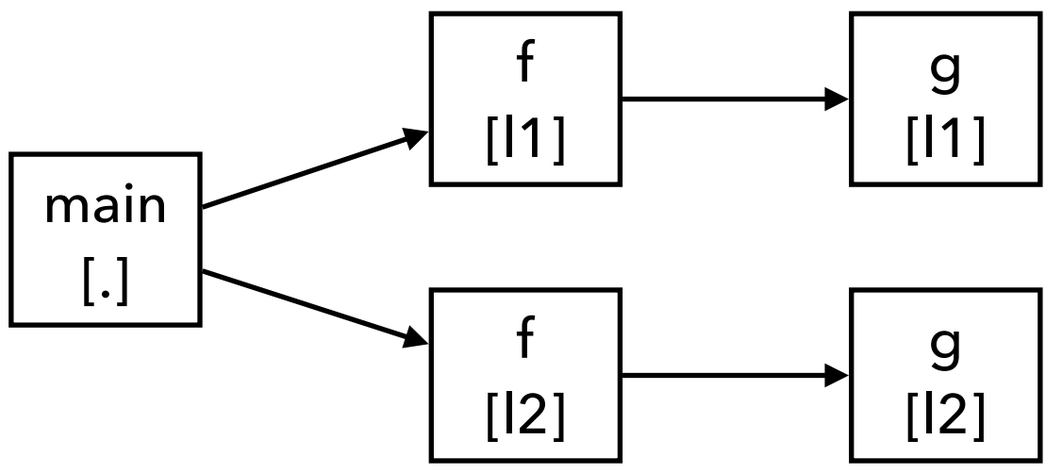
Call-site vs. Object Sensitivity

- Typical example that benefits from object sensitivity:

```
class A:
  def g(self):
     return
  def f(self):
     return self.g() // l5

def main():
   a = A()  // l1
   b = A()  // l2
   a.f()    // l3
   b.f()    // l4
```



1-call-site sensitivity



1-object sensitivity

Domains

V: the set of program variables
H: the set of allocation sites
F: the set of field names
M: the set of method identifiers
S: the set of method signatures
I: the set of instructions
T: the set of class types
$\mathbb{N}$: the set of natural numbers
C: a set of calling contexts
HC: a set of heap contexts

Output Relations

- The output relations are modified to add contexts:

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

VarPointsTo($var : V, ctx : C, heap : H, hctx : HC$)
FldPointsTo($baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC$)
InterProcAssign($to : V, toCtx : C, from : V, fromCtx : C$)
CallGraph($invo : I, callerCtx : C, meth : M, calleeCtx : C$)
Reachable($meth : M, ctx : C$)

Context Constructors

- Different choices of constructors yield different context-sensitivity flavors

$$\textbf{Record}(heap : H, ctx : C) = newHCtx : HC$$
$$\textbf{Merge}(heap : H, hctx : HC, invo : I, ctx : C) = newCtx : C$$

- **Record** generates heap contexts
- **Merge** generates calling contexts

Analysis Rules

$$\textbf{Record}(heap, ctx) = hctx,$$

$$\text{VarPointsTo}(var, ctx, heap, hctx) \leftarrow$$
$$\quad \text{Reachable}(meth, ctx), \text{Alloc}(var, heap, meth)$$

$$\text{VarPointsTo}(to, ctx, heap, hctx) \leftarrow$$
$$\quad \text{Move}(to, from), \text{VarPointsTo}(from, ctx, heap, hctx)$$

$$\text{FldPointsTo}(baseH, baseHCtx, fld, heap, hctx) \leftarrow$$
$$\quad \text{Store}(base, fld, from), \text{VarPointsTo}(from, ctx, heap, hctx),$$
$$\quad \text{VarPointsTo}(base, ctx, baseH, baseHCtx)$$

$$\text{VarPointsTo}(to, ctx, heap, hctx) \leftarrow$$
$$\quad \text{Load}(to, base, fld), \text{VarPointsTo}(base, ctx, baseH, baseHCtx),$$
$$\quad \text{FldPointsTo}(baseH, baseHCtx, fld, heap, hctx)$$

Analysis Rules

$$\textbf{Merge}(heap, hctx, invo, callerCtx) = calleeCtx,$$
$$\text{Reachable}(toMeth, calleeCtx),$$
$$\text{VarPointsTo}(this, calleeCtx, heap, hctx),$$
$$\text{CallGraph}(invo, callerCtx, toMeth, calleeCtx) \leftarrow$$
$$\text{VCall}(base, sig, invo, inMeth), \text{Reachable}(inMeth, callerCtx),$$
$$\text{VarPointsTo}(base, callerCtx, heap, hctx),$$
$$\text{HeapType}(heap, heapT), \text{LookUp}(heapT, sig, toMeth),$$
$$\text{ThisVar}(toMeth, this)$$

Analysis Rules

$$\begin{aligned}&\text{InterProcAssign}(to, calleeCtx, from, callerCtx) \leftarrow \\ &\quad \text{CallGraph}(invo, callerCtx, meth, calleeCtx), \\ &\quad \text{FormalArg}(meth, n, to), \text{ActualArg}(invo, n, from)\end{aligned}$$

$$\begin{aligned}&\text{InterProcAssign}(to, callerCtx, from, calleeCtx) \leftarrow \\ &\quad \text{CallGraph}(invo, callerCtx, meth, calleeCtx), \\ &\quad \text{FormalReturn}(meth, from), \text{ActualReturn}(invo, to)\end{aligned}$$

$$\begin{aligned}&\text{VarPointsTo}(to, toCtx, heap, hctx) \leftarrow \\ &\quad \text{InterProcAssign}(to, toCtx, from, fromCtx), \\ &\quad \text{VarPointsTo}(from, fromCtx, heap, hctx)\end{aligned}$$

Call-Site Sensitivity

- The best-known flavor of context sensitivity, which uses call-sites as contexts.
- A method is analyzed under 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, …, up to a pre-defined depth (*k*)

Call-Site Sensitivity

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

$$C = I, \qquad HC = \{ \star \}$$
$$\textbf{Record}(heap, ctx) = \star$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = invo$$

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

$$C = I, \qquad HC = I$$
$$\textbf{Record}(heap, ctx) = ctx$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = invo$$

- 2-call-site sensitivity with 1-call-site senstive heap:

$$C = I \times I, \qquad HC = I$$
$$\textbf{Record}(heap, ctx) = first(ctx)$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = pair(invo, first(ctx))$$

Object Sensitivity

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

```
class A:
   def m(self):
      return

a = A()   // l1
a.m()     // l2
```

Object Sensitivity

- 1-object sensitivity with context-insensitive heap:

$$C = H, \qquad HC = \{ \star \}$$
$$\textbf{Record}(heap, ctx) = \star$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = heap$$

- 2-object sensitivity with 1-call-site senstive heap:

$$C = H \times H, \qquad HC = H$$
$$\textbf{Record}(heap, ctx) = first(ctx)$$
$$\textbf{Merge}(heap, hctx, invo, ctx) = pair(heap, hctx)$$