AAA616: Program Analysis

Lecture 8 — Pointer Analysis

Hakjoo Oh
2022 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 $q = \&y$:
- $p = \&x$ and $q \neq \&y$:
- $p \neq \&x$ and $q = \&y$:
- $p \neq \&x$ and $q \neq \&y$:

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:

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

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

$$l \lor \neg l_1 \lor \neg l_2 \lor \cdots \lor \neg l_n \iff l \leftarrow l_1 \land l_2 \land \cdots \land l_n$$

- 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:

$$[\![P]\!] \in \wp(G)$$

  where G is the set of ground literals (literals without variables).
- A Datalog rule l :- $l_1, \ldots, l_n$ denotes the function:

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

  where σ is a variable substitution.
- The semantics of P is defined as the least fixed point of F_P:

$$[\![P]\!] = \mathit{lfp} F_P \;\text{ where } F_P(X) = X \cup \bigcup_{c \in P} f_c(X)$$

- The semantics is monotone:

$$P_1 \subseteq P_2 \implies [\![P_1]\!] \subseteq [\![P_2]\!]$$

Programs as Relations

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

$\textsc{Alloc}\ (\boldsymbol{var} : \boldsymbol{V}, \boldsymbol{heap} : \boldsymbol{H})$
$\textsc{Move}\ (\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{from} : \boldsymbol{V})$
$\textsc{Load}(\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{base} : \boldsymbol{V}, \boldsymbol{fld} : \boldsymbol{F})$
$\textsc{Store}(\boldsymbol{base} : \boldsymbol{V}, \boldsymbol{fld} : \boldsymbol{F}, \boldsymbol{from} : \boldsymbol{V})$

domains:

- $\boldsymbol{V}$ is a set of program variables
- $\boldsymbol{H}$ is a set of heap abstractions (i.e. allocation sites)
- $\boldsymbol{F}$ 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:

$\textsc{VarPointsTo}\ (\boldsymbol{var} : \boldsymbol{V}, \boldsymbol{heap} : \boldsymbol{H})$
$\textsc{FldPointsTo}\ (\boldsymbol{baseH} : \boldsymbol{H}, \boldsymbol{fld} : \boldsymbol{F}, \boldsymbol{heap} : \boldsymbol{H})$

Analysis rules:

$$\textsc{VarPointsTo}(\boldsymbol{var}, \boldsymbol{heap}) \leftarrow \textsc{Alloc}(\boldsymbol{var}, \boldsymbol{heap})$$

$$\begin{aligned}&\textsc{VarPointsTo}(\boldsymbol{to}, \boldsymbol{heap}) \leftarrow\\ &\quad \textsc{Move}(\boldsymbol{to}, \boldsymbol{from}), \textsc{VarPointsTo}(\boldsymbol{from}, \boldsymbol{heap})\end{aligned}$$

$$\begin{aligned}&\textsc{FldPointsTo}(\boldsymbol{baseH}, \boldsymbol{fld}, \boldsymbol{heap}) \leftarrow\\ &\quad \textsc{Store}(\boldsymbol{base}, \boldsymbol{fld}, \boldsymbol{from}), \textsc{VarPointsTo}(\boldsymbol{from}, \boldsymbol{heap}),\\ &\quad \textsc{VarPointsTo}(\boldsymbol{base}, \boldsymbol{baseH}).\end{aligned}$$

Exercise) Define the rule for LOAD.

Interprocedural Analysis

Domains:

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

Interprocedural Analysis (First-Order)

- Input relations:

  $\textsc{Alloc}\ (\boldsymbol{var} : \boldsymbol{V}, \boldsymbol{heap} : \boldsymbol{H}, \boldsymbol{inMeth} : \boldsymbol{M})$
  $\textsc{Move}\ (\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{from} : \boldsymbol{V})$
  $\textsc{Load}(\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{base} : \boldsymbol{V}, \boldsymbol{fld} : \boldsymbol{F})$
  $\textsc{Store}(\boldsymbol{base} : \boldsymbol{V}, \boldsymbol{fld} : \boldsymbol{F}, \boldsymbol{from} : \boldsymbol{V})$
  $\textsc{CallGraph}(\boldsymbol{invo} : \boldsymbol{I}, \boldsymbol{meth} : \boldsymbol{M})$
  $\textsc{Reachable}(\boldsymbol{meth} : \boldsymbol{M})$
  $\textsc{FormalArg}(\boldsymbol{meth} : \boldsymbol{M}, \boldsymbol{i} : \mathbb{N}, \boldsymbol{arg} : \boldsymbol{V})$
  $\textsc{ActualArg}(\boldsymbol{invo} : \boldsymbol{I}, \boldsymbol{i} : \mathbb{N}, \boldsymbol{arg} : \boldsymbol{V})$
  $\textsc{FormalReturn}(\boldsymbol{meth} : \boldsymbol{M}, \boldsymbol{ret} : \boldsymbol{V})$
  $\textsc{ActualReturn}(\boldsymbol{invo} : \boldsymbol{I}, \boldsymbol{var} : \boldsymbol{V})$

- Output relations:

  $\textsc{VarPointsTo}\ (\boldsymbol{var} : \boldsymbol{V}, \boldsymbol{heap} : \boldsymbol{H})$
  $\textsc{FldPointsTo}\ (\boldsymbol{baseH} : \boldsymbol{H}, \boldsymbol{fld} : \boldsymbol{F}, \boldsymbol{heap} : \boldsymbol{H})$
  $\textsc{InterProcAssign}\ (\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{from} : \boldsymbol{V})$

Analysis Rules

$$\textsc{VarPointsTo}(\boldsymbol{var}, \boldsymbol{heap}) \leftarrow$$
$$\quad \textsc{Reachable}(\boldsymbol{meth}), \textsc{Alloc}(\boldsymbol{var}, \boldsymbol{heap}, \boldsymbol{meth})$$

$$\textsc{VarPointsTo}(\boldsymbol{to}, \boldsymbol{heap}) \leftarrow$$
$$\quad \textsc{Move}(\boldsymbol{to}, \boldsymbol{from}), \textsc{VarPointsTo}(\boldsymbol{from}, \boldsymbol{heap})$$

$$\textsc{FldPointsTo}(\boldsymbol{baseH}, \boldsymbol{fld}, \boldsymbol{heap}) \leftarrow$$
$$\quad \textsc{Store}(\boldsymbol{base}, \boldsymbol{fld}, \boldsymbol{from}), \textsc{VarPointsTo}(\boldsymbol{from}, \boldsymbol{heap}),$$
$$\quad \textsc{VarPointsTo}(\boldsymbol{base}, \boldsymbol{baseH}).$$

$$\textsc{VarPointsTo}(\boldsymbol{to}, \boldsymbol{heap}) \leftarrow$$
$$\quad \textsc{Load}(\boldsymbol{to}, \boldsymbol{base}, \boldsymbol{fld}), \textsc{VarPointsTo}(\boldsymbol{base}, \boldsymbol{baseH}),$$
$$\quad \textsc{FldPointsTo}(\boldsymbol{baseH}, \boldsymbol{fld}, \boldsymbol{heap}).$$

$$\textsc{InterProcAssign}(\boldsymbol{to}, \boldsymbol{from}) \leftarrow \textsc{CallGraph}(\boldsymbol{invo}, \boldsymbol{meth}),$$
$$\quad \textsc{FormalArg}(\boldsymbol{meth}, \boldsymbol{n}, \boldsymbol{to}), \textsc{ActualArg}(\boldsymbol{invo}, \boldsymbol{n}, \boldsymbol{from}).$$

$$\textsc{InterProcAssign}(\boldsymbol{to}, \boldsymbol{from}) \leftarrow \textsc{CallGraph}(\boldsymbol{invo}, \boldsymbol{meth}),$$
$$\quad \textsc{FormalReturn}(\boldsymbol{meth}, \boldsymbol{from}), \textsc{ActualReturn}(\boldsymbol{invo}, \boldsymbol{to}).$$

$$\textsc{VarPointsTo}(\boldsymbol{to}, \boldsymbol{heap}) \leftarrow$$
$$\quad \textsc{InterProcAssign}(\boldsymbol{to}, \boldsymbol{from}), \textsc{VarPointsTo}(\boldsymbol{from}, \boldsymbol{heap}).$$

Example

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

Interprocedural Analysis (Higher-Order)

- Input relations:

  $\text{Alloc}\ (\boldsymbol{var} : \boldsymbol{V}, \boldsymbol{heap} : \boldsymbol{H}, \boldsymbol{inMeth} : \boldsymbol{M})$
  $\text{Move}\ (\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{from} : \boldsymbol{V})$
  $\text{Load}(\boldsymbol{to} : \boldsymbol{V}, \boldsymbol{base} : \boldsymbol{V}, \boldsymbol{fld} : \boldsymbol{F})$
  $\text{Store}(\boldsymbol{base} : \boldsymbol{V}, \boldsymbol{fld} : \boldsymbol{F}, \boldsymbol{from} : \boldsymbol{V})$
  $\text{VCall}(\boldsymbol{base} : \boldsymbol{V}, \boldsymbol{sig} : \boldsymbol{S}, \boldsymbol{invo} : \boldsymbol{I}, \boldsymbol{inMeth} : \boldsymbol{M})$
  $\text{FormalArg}(\boldsymbol{meth} : \boldsymbol{M}, \boldsymbol{i} : \mathbb{N}, \boldsymbol{arg} : \boldsymbol{V})$
  $\text{ActualArg}(\boldsymbol{invo} : \boldsymbol{I}, \boldsymbol{i} : \mathbb{N}, \boldsymbol{arg} : \boldsymbol{V})$
  $\text{FormalReturn}(\boldsymbol{meth} : \boldsymbol{M}, \boldsymbol{ret} : \boldsymbol{V})$
  $\text{ActualReturn}(\boldsymbol{invo} : \boldsymbol{I}, \boldsymbol{var} : \boldsymbol{V})$
  $\text{ThisVar}(\boldsymbol{meth} : \boldsymbol{M}, \boldsymbol{this} : \boldsymbol{V})$
  $\text{HeapType}(\boldsymbol{heap} : \boldsymbol{H}, \boldsymbol{type} : \boldsymbol{T})$
  $\text{LookUp}(\boldsymbol{type} : \boldsymbol{T}, \boldsymbol{sig} : \boldsymbol{S}, \boldsymbol{meth} : \boldsymbol{M})$

- Output relations:

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

Analysis Rules

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

$$\begin{array}{l}
\text{REACHABLE}(\boldsymbol{toMeth}), \\
\text{VARPOINTSTO}(\boldsymbol{this}, \boldsymbol{heap}), \\
\text{CALLGRAPH}(\boldsymbol{invo}, \boldsymbol{toMeth}) \leftarrow \\
\quad \text{VCALL}(\boldsymbol{base}, \boldsymbol{sig}, \boldsymbol{invo}, \boldsymbol{inMeth}), \text{REACHABLE}(\boldsymbol{inMeth}), \\
\quad \text{VARPOINTSTO}(\boldsymbol{base}, \boldsymbol{heap}), \\
\quad \text{HEAPTYPE}(\boldsymbol{heap}, \boldsymbol{heapT}), \text{LOOKUP}(\boldsymbol{heapT}, \boldsymbol{sig}, \boldsymbol{toMeth}), \\
\quad \text{THISVAR}(\boldsymbol{toMeth}, \boldsymbol{this}).
\end{array}$$

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 quality 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

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

Output Relations for Context-Sensitive Analysis

The output relations are modified to add contexts:

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

Context constructors:

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

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

Analysis Rules

$\textbf{Record}\ (heap, ctx) = hctx,$
$\textsc{VarPointsTo}(var, ctx, heap, hctx) \leftarrow$
$\textsc{Reachable}(meth, ctx), \textsc{Alloc}(var, heap, meth).$

$\textsc{VarPointsTo}(to, ctx, heap, hctx) \leftarrow$
$\textsc{Move}(to, from), \textsc{VarPointsTo}(from, ctx, heap, hctx).$

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

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

Analysis Rules

$$
\begin{aligned}
&\textbf{Merge}\ (\mathit{heap},\ \mathit{hctx},\ \mathit{invo},\ \mathit{callerCtx}) = \mathit{calleeCtx},\\
&\textsc{Reachable}(\mathit{toMeth},\ \mathit{calleeCtx}),\\
&\textsc{VarPointsTo}(\mathit{this},\ \mathit{calleeCtx},\ \mathit{heap},\ \mathit{hctx}),\\
&\textsc{CallGraph}(\mathit{invo},\ \mathit{callerCtx},\ \mathit{toMeth},\ \mathit{calleeCtx}) \leftarrow\\
&\quad \textsc{VCall}(\mathit{base},\ \mathit{sig},\ \mathit{invo},\ \mathit{inMeth}),\ \textsc{Reachable}(\mathit{inMeth},\ \mathit{callerCtx}),\\
&\quad \textsc{VarPointsTo}(\mathit{base},\ \mathit{callerCtx},\ \mathit{heap},\ \mathit{hctx}),\\
&\quad \textsc{HeapType}(\mathit{heap},\ \mathit{heapT}),\ \textsc{LookUp}(\mathit{heapT},\ \mathit{sig},\ \mathit{toMeth}),\\
&\quad \textsc{ThisVar}(\mathit{toMeth},\ \mathit{this}).
\end{aligned}
$$

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

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

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

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:

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

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

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

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

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

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:

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

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

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

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.