Pointer Analysis
(but really: value-flow analysis)

- What objects can a variable point to?

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

foo:a	new A1()
bar:a	new A2()

objects represented by allocation sites

Pointer Analysis

- What objects can a variable point to?

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

foo:a	new A1()
bar:a	new A2()
id:a	new A1(), new A2()

Pointer Analysis

- What objects can a variable point to?

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

foo:a	new A1()
bar:a	new A2()
id:a	new A1(), new A2()
foo:b	new A1(), new A2()
bar:b	new A1(), new A2()

context-sensitive points-to

foo:a	new A1()
bar:a	new A2()
id:a (foo)	new A1()
id:a (bar)	new A2()
foo:b	new A1()
bar:b	new A2()

Datalog To The Rescue!

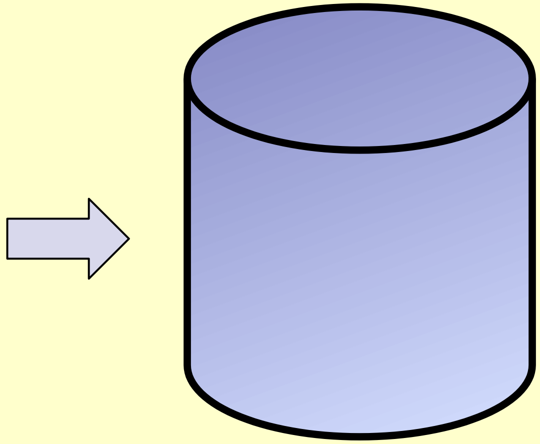
- Datalog is relations + recursion
- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - e.g., as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - monotonic

Less programming, more specification

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```



Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

rules

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

head

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

head relation

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

bodies

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

body relations

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

join variable

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

recursion

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()

1st rule result

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()

2nd rule evaluation

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()
a	new B()

2nd rule result

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Datalog: Declarative Mutual Recursion

source
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;

Alloc

a	new A()
b	new B()
c	new C()

Move

a	b
b	a
c	b

VarPointsTo

a	new A()
b	new B()
c	new C()
a	new B()
b	new A()
c	new B()
c	new A()

```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

VarPointsTo(to, obj) <-
  Move(to, from),
  VarPointsTo(from, obj).
```

Expressiveness and Insights

- Greatest benefit of the declarative approach: better algorithms
  - the same algorithms can be described non-declaratively
    - the algorithms are interesting regardless of *how* they are implemented
  - but the declarative formulation was helpful in finding them
    - and in conjecturing that they work well

Recall: Context-Sensitivity (call-site sensitivity)

- What objects can a variable point to?

program

```
void foo() {
  Object a = new A1();
  Object b = id(a);
}

void bar() {
  Object a = new A2();
  Object b = id(a);
}

Object id(Object a) {
  return a;
}
```

points-to

foo:a	new A1()
bar:a	new A2()
id:a	new A1(), new A2()
foo:b	new A1(), new A2()
bar:b	new A1(), new A2()

call-site-sensitive points-to

foo:a	new A1()
bar:a	new A2()
id:a (foo)	new A1()
id:a (bar)	new A2()
foo:b	new A1()
bar:b	new A2()

Object-Sensitivity
(vs. call-site sensitivity)

program

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

1-call-site-sensitive points-to

fun1:a1	new A1()
fun2:a2	new A2()
id2:a (fun1)	new A1()
id2:a (fun2)	new A2()
id:a (id2)	new A1(),new A2()
id2:ret (*)	new A1(),new A2()
fun1:b1	new A1(),new A2()
fun2:b2	new A1(),new A2()

Object-Sensitivity

program

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

1-object-sensitive points-to

fun1:a1	new A1()
fun2:a2	new A2()
id2:a (C*1*)	new A1()
id2:a (D*1*)	new A2()
id:a (C*1*)	new A1()
id:a (D*1*)	new A2()
id2:ret (C*1*)	new A1()
fun1:b1	new A1()
fun2:b2	new A2()

A General Formulation of Context-Sensitive Analyses

- *Every context-sensitive flow-insensitive analysis there is* (ECSFIATI)
  - ok, almost every
    - most not handled are strictly less sophisticated
  - and also many more than people ever thought
- Also with on-the-fly call-graph construction
- In 9 easy rules!

Simple Intermediate Language

- We consider Java-bytecode-like language
  - allocation instructions (`Alloc`)
  - local assignments (`Move`)
  - virtual and static calls (`VCall`, `SCall`)
  - field access, assignments (`Load`, `Store`)
  - standard type system and symbol table info (`Type`, `Subtype`, `FormalArg`, `ActualArg`, etc.)

Rule 1: Allocating Objects
(Alloc)

```
Record(obj, ctx) = hctx,
VarPointsTo(var, ctx, obj, hctx)
<-
  Alloc(var, obj, meth),
  Reachable(meth, ctx).
```

obj: var = new Something();

Rule 2: Variable Assignment

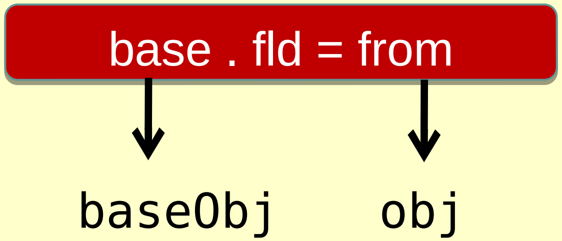
(Move)

```
VarPointsTo(to, ctx, obj, hctx)
<-
  Move(to, from),
  VarPointsTo(from, ctx, obj, hctx).
```

to = from

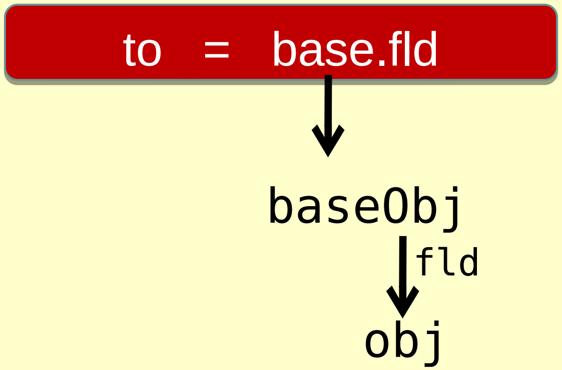
Rule 3: Object Field Write (Store)

```
FldPointsTo(baseObj, baseHCtx, fld, obj, hctx)
<-
  Store(base, fld, from),
  VarPointsTo(from, ctx, obj, hctx),
  VarPointsTo(base, ctx, baseObj, baseHCtx).
```



Rule 4: Object Field Read (Load)

```
VarPointsTo(to, ctx, obj, hctx)
<-
  Load(to, base, fld),
  FldPointsTo(baseObj, baseHCtx, fld, obj, hctx),
  VarPointsTo(base, ctx, baseObj, baseHCtx).
```



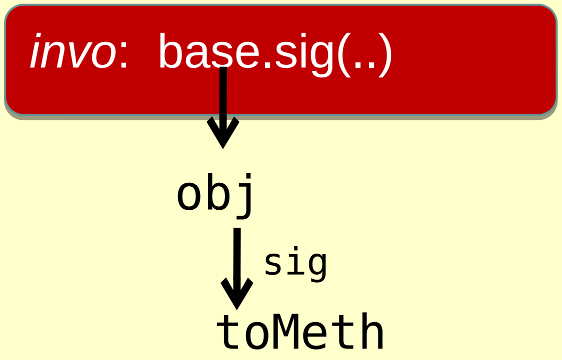
Rule 5: Static Method Calls
(SCall)

```
MergeStatic(invo, callerCtx) = calleeCtx,
Reachable(toMeth, calleeCtx),
CallGraph(invo, callerCtx, toMeth, calleeCtx)
<-
  SCall(toMeth, invo, inMeth),
  Reachable(inMeth, callerCtx).
```

invo: toMeth(..)

Rule 6: Virtual Method Calls
(VCall)

```
Merge(obj, hctx, invo, callerCtx) = calleeCtx,
Reachable(toMeth, calleeCtx),
VarPointsTo(this, calleeCtx, obj, hctx),
CallGraph(invo, callerCtx, toMeth, calleeCtx)
<-
  VCall(base, sig, invo, inMeth),
  Reachable(inMeth, callerCtx),
  VarPointsTo(base, callerCtx, obj, hctx),
  LookUp(obj, sig, toMeth),
  ThisVar(toMeth, this).
```



Rule 7: Parameter Passing

```
InterProcAssign(to, calleeCtx, from, callerCtx)
<-
  CallGraph(invo, callerCtx, meth, calleeCtx),
  ActualArg(invo, i, from),
  FormalArg(meth, i,to).
```

invo: meth(.., from, ..) --> meth(.., to, ..)

Rule 8: Return Value Passing

```
InterProcAssign(to, callerCtx, from, calleeCtx)
<-
  CallGraph(invo, callerCtx, meth, calleeCtx),
  ActualReturn(invo, to),
  FormalReturn(meth, from).
```

invo: to = meth(..) --> meth(..) { .. return from; }

Rule 9: Parameter/Result Passing as Assignment

```
VarPointsTo(to, toCtx, obj, hctx)
<-
  InterProcAssign(to, toCtx, from, fromCtx),
  VarPointsTo(from, fromCtx, obj, hctx).
```

Can Now Express Past Analyses Nicely

- 1-call-site-sensitive with context-sensitive heap:
  - *Context* = *HContext* = Instr
- Functions:
  - ***Record***`(obj, ctx) = ctx`
  - ***Merge***`(obj, hctx, invo, callerCtx) = invo`
  - ***MergeStatic***`(invo, callerCtx) = invo`

Can Now Express Past Analyses Nicely

- 1-object-sensitive+heap:
  - *Context* = *HContext* = Instr
- Functions:
  - ***Record***`(obj, ctx) = ctx`
  - ***Merge***`(obj, hctx, invo, callerCtx) = obj`
  - ***MergeStatic***`(invo, callerCtx) = callerCtx`

Can Now Express Past Analyses Nicely

- PADDLE-style 2-object-sensitive+heap:
  - *Context* = Instr^2 , *HContext* = Instr
- Functions:
  - ***Record***(obj, ctx) = first(ctx)
  - ***Merge***(obj, hctx, invo, callerCtx) = pair(obj, first(ctx))
  - ***MergeStatic***(invo, callerCtx) = callerCtx

Lots of Insights and New Algorithms (all with major benefits)

- Discovered that the same name was used for two past algorithms with very different behavior
- Proposed a new kind of context (*type-sensitivity*), easily implemented by uniformly tweaking **`Record`**/**`Merge`** functions
- Found connections between analyses in functional/OO languages
- Showed that merging different kinds of contexts works great (*hybrid context-sensitivity*)