what's pointer analysis? 1

- determine where a variable might point (refer) to

why?

- optimization
- bug detection and security
- software understanding

$\Rightarrow$ mostly as a service to other program analysis

$\Rightarrow$ fundamental topic in program analysis

program

```
X: a = new Object();
Y: b = new Object();
Z: c = new Object();
   a = b;
   b = a;
   c = b;
```

facts

```
AssignHeapAllocation("a", "X").
AssignHeapAllocation("b", "Y").
AssignHeapAllocation("c", "Z").
AssignLocal("b", "a").
AssignLocal("a", "b").
AssignLocal("b", "c").
```

straight-line pointer analysis in datalog (1)

program

```
X: a = new Object();
Y: b = new Object();
Z: c = new Object();
   a = b;
   b = a;
   c = b;
```

```
AssignHeapAllocation(?heap, ?to) ->
  HeapAllocationRef(?heap),
  VarRef(?to).
```

facts

```
AssignHeapAllocation("a", "X").
AssignHeapAllocation("b", "Y").
AssignHeapAllocation("c", "Z").
AssignLocal("b", "a").
AssignLocal("a", "b").
AssignLocal("b", "c").
```

```
AssignLocal(?from, ?to) ->
  VarRef(?from),
  VarRef(?to).
```

datalog

```
VarPointsTo(?var, ?heap) <-
  AssignHeapAllocation(?heap, ?var).

VarPointsTo(?to, ?heap) <-
  AssignLocal(?from, ?to),
  VarPointsTo(?from, ?heap).
```

query

```
?- VarPointsTo(?var, ?heap).
  ("a", "X") ("a", "Y")
  ("b", "X") ("b", "Y")
  ("c", "X") ("c", "Y") ("c", "Z")
```

inter-procedural pointer analysis in datalog (1)


```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

```
FormalParam["f", 0] = "o".
ReturnVar("f", "o").

AssignHeapAllocation("a", "X").
CallGraphEdge("main/f", "f").
ActualParam["main/f", 0] = "a".
AssignReturnValue("main/f", "b").
```

inter-procedural pointer analysis in datalog (1)


```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

```
FormalParam[?method, ?index] = ?var ->
  MethodDeclarationRef(?method),
  ParamIndexRef(?index),
  VarRef(?var).
```

```
FormalParam["f", 0] = "o".
ReturnVar("f", "o").

AssignHeapAllocation("a", "X").
CallGraphEdge("main/f", "f").
ActualParam["main/f", 0] = "a".
AssignReturnValue("main/f", "b").
```

inter-procedural pointer analysis in datalog (1)


```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

```
ReturnVar(?method, ?var) ->
  MethodDeclarationRef(?method),
  VarRef(?var).
```

```
FormalParam["f", 0] = "o".
ReturnVar("f", "o").

AssignHeapAllocation("a", "X").
CallGraphEdge("main/f", "f").
ActualParam["main/f", 0] = "a".
AssignReturnValue("main/f", "b").
```

inter-procedural pointer analysis in datalog (1)


```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

```
CallGraphEdge(?invocation, ?method) ->
  MethodInvocationRef(?invocation),
  MethodDeclarationRef(?method).
```

```
FormalParam["f", 0] = "o".
ReturnVar("f", "o").

AssignHeapAllocation("a", "X").
CallGraphEdge("main/f", "f").
ActualParam["main/f", 0] = "a".
AssignReturnValue("main/f", "b").
```

inter-procedural pointer analysis in datalog (1)


```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

```
ActualParam[?invocation, ?index] = ?var ->
  MethodInvocationRef(?invocation),
  ParamIndexRef(?index),
  VarRef(?var).
```

```
FormalParam["f", 0] = "o".
ReturnVar("f", "o").

AssignHeapAllocation("a", "X").
CallGraphEdge("main/f", "f").
ActualParam["main/f", 0] = "a".
AssignReturnValue("main/f", "b").
```

inter-procedural pointer analysis in datalog (1)


```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

```
AssignReturnValue(?invocation, ?to) ->
  MethodInvocationRef(?invocation),
  VarRef(?to).
```

```
FormalParam["f", 0] = "o".
ReturnVar("f", "o").

AssignHeapAllocation("a", "X").
CallGraphEdge("main/f", "f").
ActualParam["main/f", 0] = "a".
AssignReturnValue("main/f", "b").
```

inter-procedural pointer analysis in datalog (2)


```
VarPointsTo(?var, ?heap) <-
  AssignHeapAllocation(?heap, ?var).

VarPointsTo(?to, ?heap) <-
  Assign(?from, ?to), VarPointsTo(?from, ?heap).

Assign(?from, ?to) <- AssignLocal(?from, ?to).

Assign(?actual, ?formal) <-
  FormalParam[?method, ?index] = ?formal,
  ActualParam[?invocation, ?index] = ?actual,
  CallGraphEdge(?invocation, ?method).

Assign(?return, ?local) <-
  ReturnVar(?method, ?return),
  CallGraphEdge(?invocation, ?method),
  AssignReturnValue(?invocation, ?local).
```

program

```
void main() {
  X: a = new Object();
     b = f(a);
}

void f(Object o) {
  return o;
}
```

points-to information

```
?- VarPointsTo(?var, ?heap).
  ("a", "X")
  ("o", "X")
  ("b", "X")
```

- directionality
- flow sensitivity
- call graph precision
- reachability [skip]
- context sensitivity
- heap abstraction
- field abstraction [skip]

subset constraints

- a = b → VarPointsTo("b") ⊆ VarPointsTo("a")

```
VarPointsTo(?to, ?heap) <-
  Assign(?from, ?to), VarPointsTo(?from, ?heap).
```

equivalence constraint

- a = b → VarPointsTo("a") = VarPointsTo("b")

```
VarPointsTo(?to, ?heap) <-
  Assign(?from, ?to), VarPointsTo(?from, ?heap).

VarPointsTo(from, ?heap) <-
  Assign(?from, ?to), VarPointsTo(?to, ?heap).
```

```
X: a = new Object();
   b = a;
Y: a = new Object();
```

```
AssignHeapAllocation("a", "X").
AssignHeapAllocation("a", "Y").
AssignLocal("a", "b").
```

```
?- VarPointsTo(?var, ?heap).
  ("a", "X") ("a", "Y") ("b", "X") ("b", "Y")
```

- flow-sensitive analysis is unpopular
- most of the benefits can be achieved with SSA

```
void main() {
  Base a = new Extension1();
  a.f();
  Base b = new Extension2();
  b.f();
}

class Base {
  void f() {}
}

class Extension1 extends Base {
  void f() {}
}

class Extension2 extends Base {
}
```

pre-computed

- CHA
- RTA
- imprecise pointer analysis (e.g. Whaley)

```
CallGraphEdge("main.f1", "Base.f").
CallGraphEdge("main.f1", "Extension1.f").
CallGraphEdge("main.f1", "Extension2.f").

CallGraphEdge("main.f2", "Base.f").
CallGraphEdge("main.f2", "Extension1.f").
CallGraphEdge("main.f2", "Extension2.f").
```

better: **on-the-fly** using virtual method resolution

on-the-fly call graph computation (1)


```
AssignHeapAllocation("a", "new Extension1").
AssignHeapAllocation("b", "new Extension2").

HeapAllocation("new Extension1", "Extension1").
HeapAllocation("new Extension2", "Extension2").

VirtualMethodInvocation("main.f1", "a", "<Base: void f()>").
VirtualMethodInvocation("main.f2", "b", "<Base: void f()>").

MethodSignarure("<Base: void f()>", "Base", "f", "void()").
MethodSignarure("<Extension1: void f()>", "Extension1", "f", "void()").

MethodDeclaration["<Base: void f()>"] = "Base.f".
MethodDeclaration["<Extension1: void f()>"] = "Extension1.f".

SuperType("Base", "java.lang.Object").
SuperType("Extension1", "Base").
SuperType("Extension2", "Base").
```

on-the-fly call graph computation (1)


```
AssignH
AssignH

HeapAll
HeapAll

VirtualMethodInvocation("main.f1", "a", "<Base: void f()>").
VirtualMethodInvocation("main.f2", "b", "<Base: void f()>").

MethodSignarure("<Base: void f()>", "Base", "f", "void()").
MethodSignarure("<Extension1: void f()>", "Extension1", "f", "void()").

MethodDeclaration["<Base: void f()>"] = "Base.f".
MethodDeclaration["<Extension1: void f()>"] = "Extension1.f".

SuperType("Base", "java.lang.Object").
SuperType("Extension1", "Base").
SuperType("Extension2", "Base").
```

```
VirtualMethodInvocation(?invocation, ?base, ?sig) ->
  MethodInvocationRef(?invocation),
  VarRef(?base),
  MethodSignatureRef(?sig).
```

on-the-fly call graph computation (1)


```
AssignHeapAllocation("a", "new Extension1").
AssignHeapAllocation("b", "new Extension2").

HeapAllocation("new Extension1", "Extension1").
HeapAllocation("new Extension2", "Extension2").

VirtualMethodInvocation("main.f1", "a", "<Base: void f()>").
VirtualMethodInvocation("main.f2", "b", "<Base: void f()>").

MethodSignarure("<Base: void f()>", "Base", "f", "void()").
MethodSignarure("<Extension1: void f()>", "Extension1", "f", "void()").

Me
Me

Su
Su
Su
```

```
MethodSignature(?ref, ?type, ?simplename, ?descriptor) ->
  MethodSignatureRef(?ref),
  TypeRef(?type),
  SimpleNameRef(?simplename),
  MethodDescriptorRef(?descriptor).
```

on-the-fly call graph computation (1)


```
AssignHeapAllocation("a", "new Extension1").
AssignHeapAllocation("b", "new Extension2").

HeapAllocation("new Extension1", "Extension1").
HeapAllocation("new Extension2", "Extension2").

VirtualMethodInvocation("main.f1", "a", "<Base: void f()>").
VirtualMethodInvocation("main.f2", "b", "<Base: void f()>").

MethodSignarure("<Base: void f()>", "Base", "f", "void()").
MethodSignarure("<Extension1: void f()>", "Extension1", "f", "void()").

MethodDeclaration["<Base: void f()>"] = "Base.f".
MethodDeclaration["<Extension1: void f()>"] = "Extension1.f".

SuperType("Base", "ja
SuperType("Extension1
SuperType("Extension2
```

```
MethodDeclaration[?signature] = ?ref ->
  MethodSignatureRef(?signature),
  MethodDeclarationRef(?ref).
```

on-the-fly call graph computation (1)


```
AssignHeapAllocation("a", "new Extension1").
AssignHeapAllocation("b", "new Extension2").

HeapAllocation("new Extension1", "Extension1").
HeapAllocation("new Extension2", "Extension2").

VirtualMethodInvocation("main.f1", "a", "<Base: void f()>").
VirtualMethodInvocation("main.f2", "b", "<Base: void f()>").

MethodSignarure("<Base: void f()>", "Base", "f", "void()").
MethodSignarure("<Extension1: void f()>", "Extension1", "f", "void()").

MethodDeclaration["<Base: void f()>
MethodDeclaration["<Extension1: voi

SuperType("Base", "java.lang.Object").
SuperType("Extension1", "Base").
SuperType("Extension2", "Base").
```

```
SuperType[?ref] = ?super ->
  TypeRef(?ref),
  TypeRef(?super).
```

virtual method resolution

```
ResolveMethod[?type, ?simplename, ?descriptor] = ?method <-
  MethodDeclared[?type, ?simplename, ?descriptor] = ?method.

ResolveMethod[?type, ?simplename, ?descriptor] = ?method <-
   SuperType[?type] = ?supertype,
   ResolveMethod[?supertype, ?simplename, ?descriptor] = ?method,
   not MethodDeclared[?type, ?simplename, ?descriptor] = _.

MethodDeclared[?type, ?simplename, ?descriptor] = ?method <-
  MethodSignature(?signature, ?type, ?simplename, ?descriptor),
  MethodDeclaration[?signature] = ?method.
```

virtual method resolution

```
CallGraphEdge(?invocation, ?tomethod) <-
  VirtualMethodInvocation(?invocation, ?base, ?signature),
  VarPointsTo(?base, ?heap),
  HeapAllocation(?heap, ?type),
  MethodSignature(?signature, _, ?simplename, ?descriptor),
  ResolveMethod[?type, ?simplename, ?descriptor] = ?tomethod.

CallGraphEdge(?invocation, ?method) <-
  StaticMethodInvocation(_, ?invocation, ?sig),
  MethodDeclaration[?sig] = ?method.
```

```
void main() {
  Base a = new Extension1();
  a.f();
  Base b = new Extension2();
  b.f();
}

class Base {
  void f() {}
}

class Extension1 extends Base {
  void f() {}
}

class Extension2 extends Base {
}
```

on-the-fly call graph computation (3)


```
?- ResolveMethod[?type, ?name, ?descriptor] = ?method.
('Base', 'f', 'void()', 'Base.f')
('Extension1', 'f', 'void()', 'Extension1.f')
('Extension2', 'f', 'void()', 'Base.f')

?- CallGraphEdge(?inv, ?m).
('main.f1', 'Extension1.f')
('main.f2', 'Base.f')
```

```
void m
  Base
  a.f(
  Base
  b.f(
}

class Base {
  void f() {}
}

class Extension1 extends Base {
  void f() {}
}

class Extension2 extends Base {
}
```

```
void main () {
  X: a = new Object();
  Y: b = new Object();
     c = identity(a);
     d = identity(b);
}
Object identity(Object o) { return o; }
```

```
CallGraphEdge("main/identity1", "identity").
CallGraphEdge("main/identity2", "identity").
ActualParam["main/identity1", 0] = "a".
ActualParam["main/identity2", 0] = "b".
AssignReturnValue("main/identity1", "c").
AssignReturnValue("main/identity2", "d").
FormalParam["identity", 0] = "o".
ReturnVar("identity", "o").
```

```
void main () {
  X: a = new Object();
  Y: b = new Object();
     c = identity(a);
     d = identity(b);
}
Object identity(Object o) { return o; }
```

```
CallGraphEdge("main/identity1", "identity").
CallGraphEdge("main/identity2", "identity").
ActualParam["main/identity1", 0] = "a".
ActualParam["main/identity2", 0] = "b".
AssignReturnValue("main/identity1", "c").
AssignReturnValue("main/identity2", "d").
FormalParam["identity", 0] = "o".
ReturnVar("identity", "o").
```

```
("a", "X")
("b", "Y")

("o", "X")
("o", "Y")

("c", "X")
("c", "Y")

("d", "X")
("d", "Y")
```

- analyse a method for multiple contexts
- context: static abstraction of a set of run-time invocations
- examples:
  - call-site
  - series of call-sites (1, 2, ..., acyclic, ∞)
  - target object
  - series of target objects
  - types of parameters
- main source of complexity
- flurry of research last decade
- not really shown to scale, in particular in combination with other essential features.

context-insensitive

```
VarPointsTo(?var, ?heap) <-
  AssignHeapAllocation(_, ?heap, ?var).
```

context-sensitive

```
VarPointsTo(?ctx, ?var, ?heap) <-
  AssignHeapAllocation(?inmethod, ?heap, ?var),
  Reachable(?ctx, ?inmethod).
```

new facts

```
AssignHeapAllocation("main", "a", "X").
AssignHeapAllocation("main", "b", "Y").
```

context-insensitive call graph

```
CallGraphEdge(?invocation, ?method)
```

context-sensitive call graph

```
CallGraphEdge(?callerCtx, ?invocation, ?invocation, ?tomethod) <-
  VirtualMethodInvocation(?inmethod, ?invocation, ?base, ?signature),
  Reachable(?callerCtx, ?inmethod),
  VarPointsTo(?callerCtx, ?base, ?heap),
  HeapAllocation(?heap, ?type),
  MethodSignature(?signature, _, ?simplename, ?descriptor),
  ResolveMethod[?type, ?simplename, ?descriptor] = ?tomethod.
```

new facts

```
MethodInvocation("main/identity1", "main", "@this", "identity").
MethodInvocation("main/identity2", "main", "@this", "identity").
```

reachability

```
Reachable(ctx, ?method) <-
  MethodInvocationRef(ctx),
  MethodInvocationRef:Value(ctx:"<<initial-context>>"),
  MainMethodDeclaration(?method).

Reachable(?ctx, ?method) <-
  CallGraphEdge(_, _, ?ctx, ?method).

MainMethodDeclaration(?method) <-
  MethodSignature(?signature, _, simplename, descriptor),
  SimpleNameRef:Value(simplename:"main"),
  MethodDescriptorRef:Value(descriptor:"void(java.lang.String[])"),
  MethodDeclaration[?signature] = ?method.
```

assignments become context-specific

```
Assign(?callerCtx, ?base, ?calleeCtx, ?this) <-
  ThisVar[?method] = ?this,
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?method),
  Base[?invocation] = ?base.

Assign(?callerCtx, ?actual, ?calleeCtx, ?formal) <-
  FormalParam[?method, ?index] = ?formal,
  ActualParam[?invocation, ?index] = ?actual,
  CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?method).

VarPointsTo(?toCtx, ?to, ?h) <-
  Assign(?fromCtx, ?from, ?toCtx, ?to),
  VarPointsTo(?fromCtx, ?from, ?h).
```

context-sensitive pointer analysis in datalog (5)


```
void main () {
  X: a = new Object();
  Y: b = new Object();
     c = identity(a);
     d = identity(b);
}

Object identity(Object o) { return o; }
```

```
?- VarPointsTo(?ctx, ?var, ?heap).
  ("<<initial-context>>", "a", "X")
  ("<<initial-context>>", "b", "Y")
  ("<<initial-context>>", "c", "X")
  ("<<initial-context>>", "d", "Y")
  ("main/identity1",  "o", "X")
  ("main/identity2",  "o", "Y")
```

how to represent an heap allocation?

```
void main() {
  a = allocate();
  b = allocate();
}

Object allocate() {
  X: o = new Object();
     return o;
}
```

- common in recent frameworks, libraries
- solution: context-sensitive heap abstraction
  - a.k.a. heap cloning
  - a.k.a. heap specialization

context-insensitive heap abstraction

```
VarPointsTo(?ctx, ?var, ?heap) :-
  AssignHeapAllocation(?inmethod, ?var, ?heap),
  Reachable(?ctx, ?inmethod).
```

context-sensitive heap abstraction

```
VarPointsTo(?ctx, ?var, Heap(?ctx, ?heap)) :-
  AssignHeapAllocation(?inmethod, ?var, ?heap),
  Reachable(?ctx, ?inmethod).
```

context-sensitive heap abstraction in datalog (2)


```
void main() {
  a = allocate();
  b = allocate();
}

Object allocate() {
  X: o = new Object();
     return o;
}
```

```
?- VarPointsTo(?ctx, ?var, ?heap).
  ("main/allocate1", "o",  Heap("main/allocate1","X"))
  ("main/allocate2", "o",  Heap("main/allocate2","X"))
  ("initial-context", "a", Heap("main/allocate1","X"))
  ("initial-context", "b", Heap("main/allocate2","X"))
```

```
InstanceFieldPointsTo(?baseheap, ?sig, ?heap) <-
  StoreInstanceField(?inmethod, ?from, ?base, ?sig),
  Reachable(?ctx, ?inmethod),
  VarPointsTo(?ctx, ?from, ?heap),
  VarPointsTo(?ctx, ?base, ?baseheap).

VarPointsTo(?ctx, ?to, ?heap) <-
  LoadInstanceField(?inmethod, ?base, ?sig, ?to),
  Reachable(?ctx, ?inmethod),
  VarPointsTo(?ctx, ?base, ?baseheap),
  InstanceFieldPointsTo(?baseheap, ?sig, ?heap).
```

pointer analysis that is:

- scalable (millions lines of code)
- fast in absolute time (seconds)

nobody has succeeded at creating an algorithm that supports:

- inclusion constraints
- on-the-fly call graph construction
- context-sensitive analysis
- context-sensitive heap abstraction

all current publications sacrifice at one, or usually more, points