Declarative Static Program Analysis with Doop

Jonathan Aldrich
17-355/17-665/17-819: Program Analysis

Slides adapted by permission from Yannis Smaragdakis

Based on work with Martin Bravenboer,
George Kastrinis,
George Balatsouras

European Research Council
Established by the European Commission

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS, CULTURE & SPORTS
MANAGING AUTHORITY
NSRF 2007-2013
EUROPEAN SOCIAL FUND
Co-financed by Greece and the European Union

LOGICBLOX

Overview

- Declarative pointer analysis
  - Specified as logical rules in Datalog
  - Supports various forms of context-sensitivity
- Efficient implementation
  - Datalog execution model
  - Optimization of Datalog queries
- Why do we care?
  - Easy to understand, modify, and optimize
  - Best analysis performance available today

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

objects represented by allocation sites

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

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	
foo:b	
bar:b	

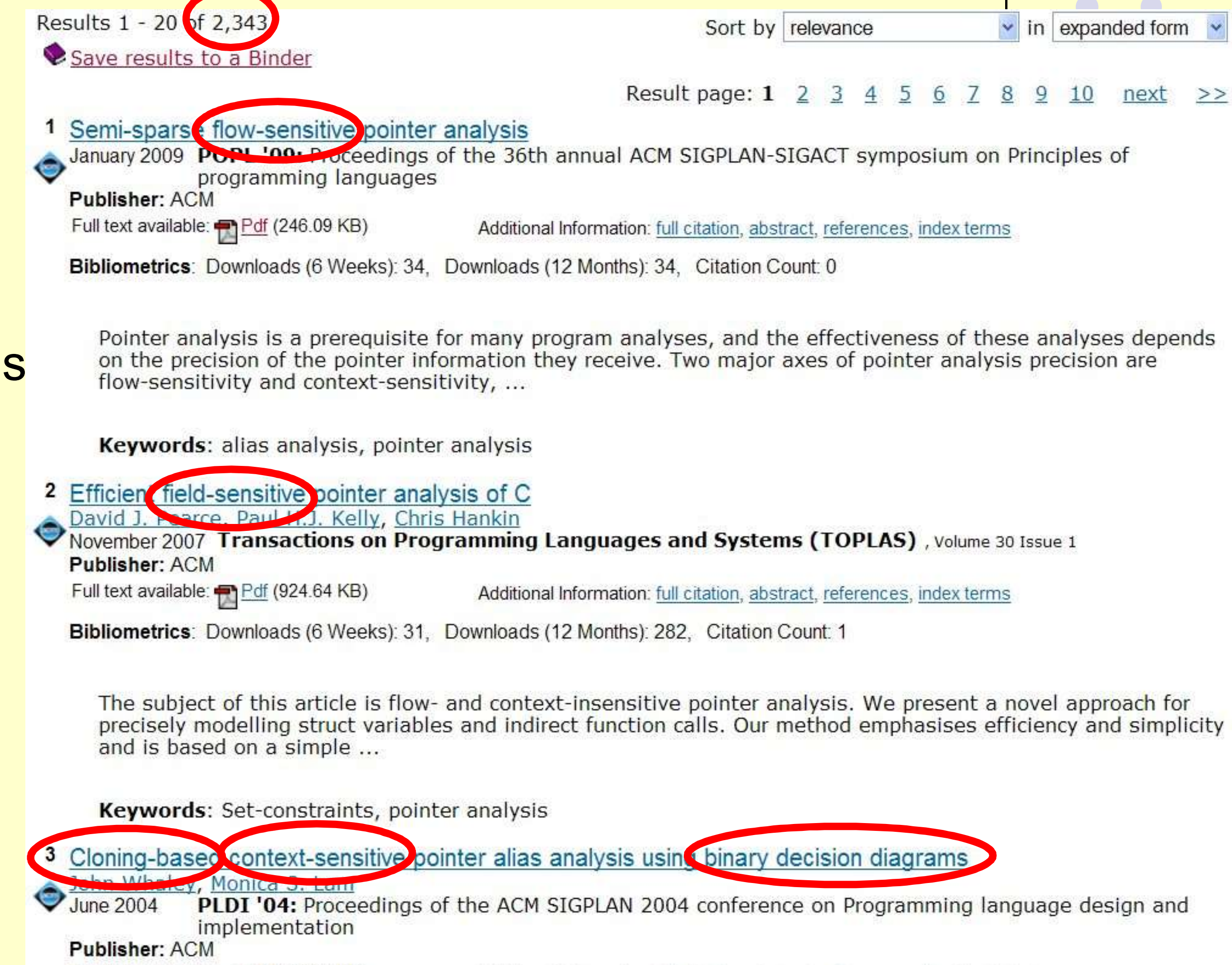
remember for later: context-sensitivity is what makes an analysis precise

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()

Pointer Analysis: A Complex Domain

- flow-sensitive
- field-sensitive
- heap cloning
- context-sensitive
- binary decision diagrams
- inclusion-based
- unification-based
- on-the-fly call graph
- k-cfa
- object sensitive
- field-based
- demand-driven


Results 1 - 20 of 2,343 Sort by relevance in expanded form

Save results to a Binder

Result page: 1 2 3 4 5 6 7 8 9 10 next >>

1 Semi-sparse flow-sensitive pointer analysis
January 2009 POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
Publisher: ACM
Full text available: Pdf (246.09 KB) Additional Information: full citation, abstract, references, index terms
Bibliometrics: Downloads (6 Weeks): 34, Downloads (12 Months): 34, Citation Count: 0

Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity, ...

Keywords: alias analysis, pointer analysis

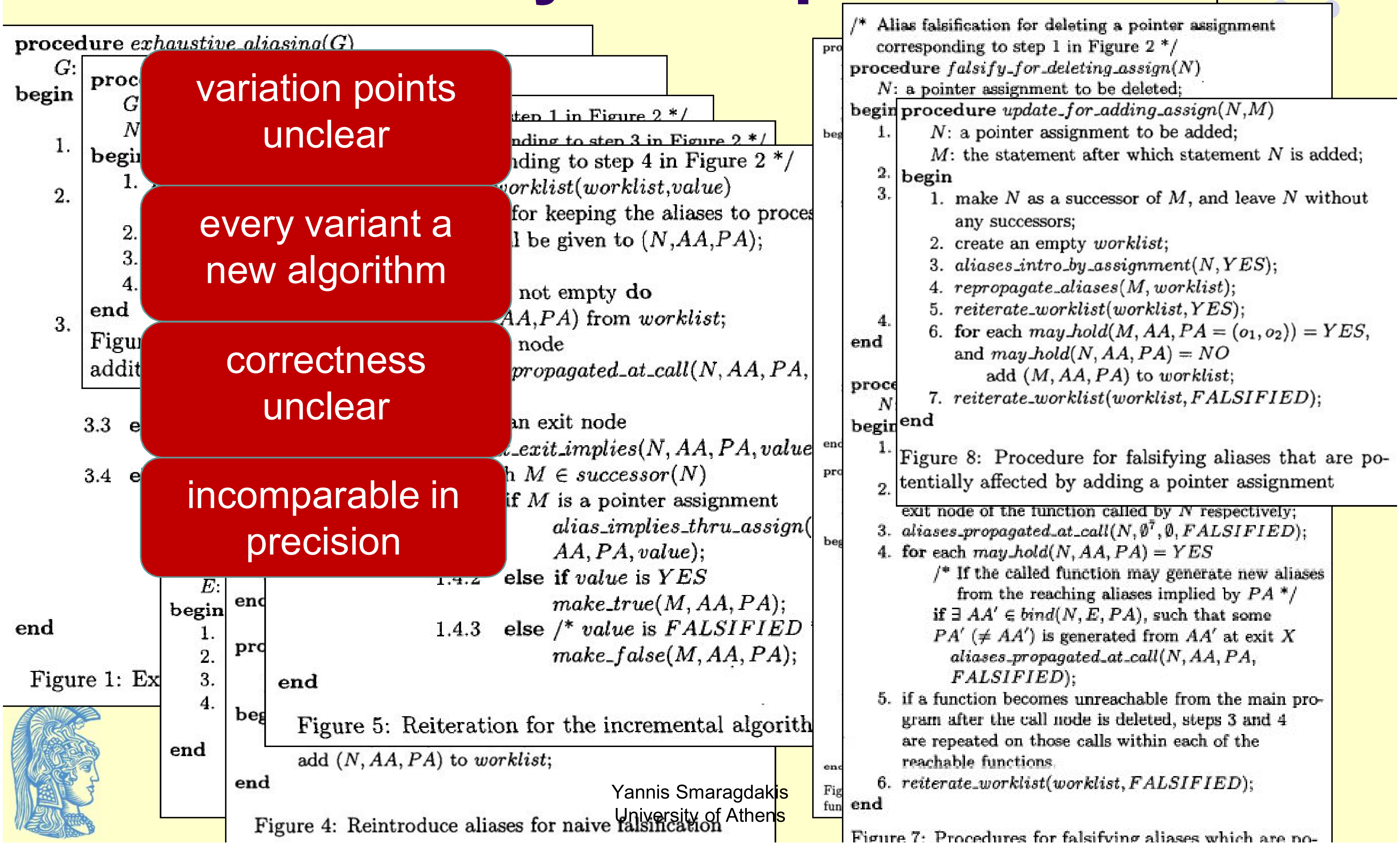
2 Efficient field-sensitive pointer analysis of C
David J. Pearce, Paul H.J. Kelly, Chris Hankin
November 2007 **Transactions on Programming Languages and Systems (TOPLAS)** , Volume 30 Issue 1
Publisher: ACM
Full text available: Pdf (924.64 KB) Additional Information: full citation, abstract, references, index terms
Bibliometrics: Downloads (6 Weeks): 31, Downloads (12 Months): 282, Citation Count: 1

The subject of this article is flow- and context-insensitive pointer analysis. We present a novel approach for precisely modelling struct variables and indirect function calls. Our method emphasises efficiency and simplicity and is based on a simple ...

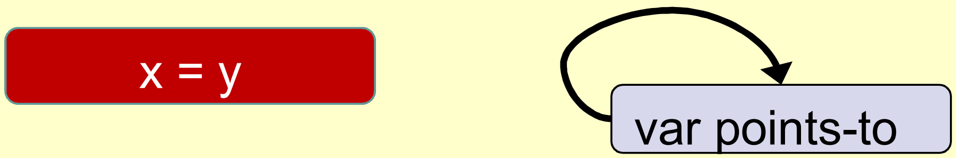
Keywords: Set-constraints, pointer analysis

3 Cloning-based context-sensitive pointer alias analysis using binary decision diagrams
John Whaley, Monica S. Lam
June 2004 **PLDI '04:** Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation
Publisher: ACM

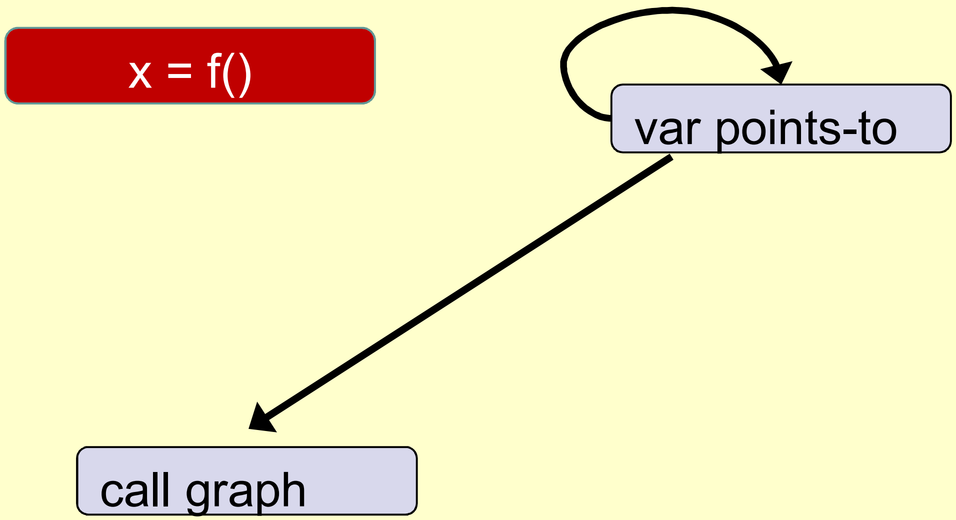
Algorithms Found In a 10-Page Pointer Analysis Paper



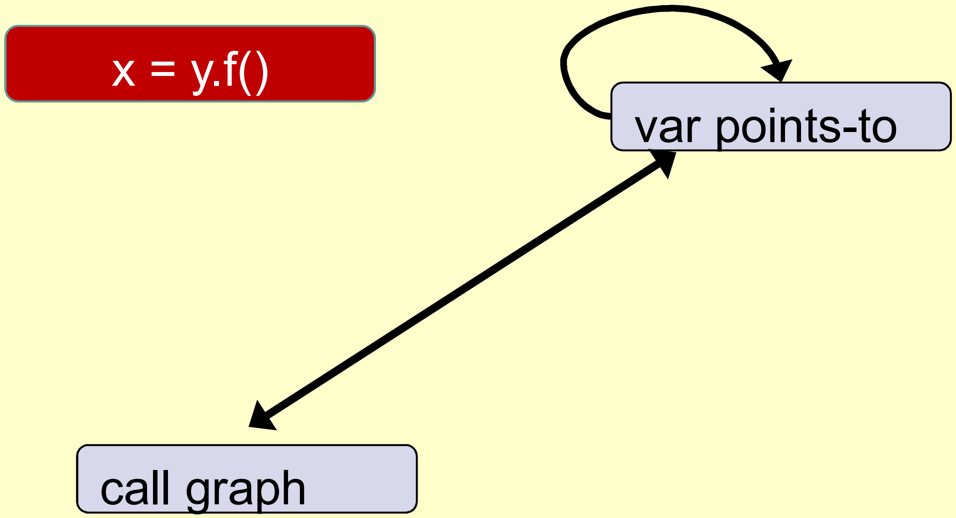
Program Analysis: a Domain of Mutual Recursion



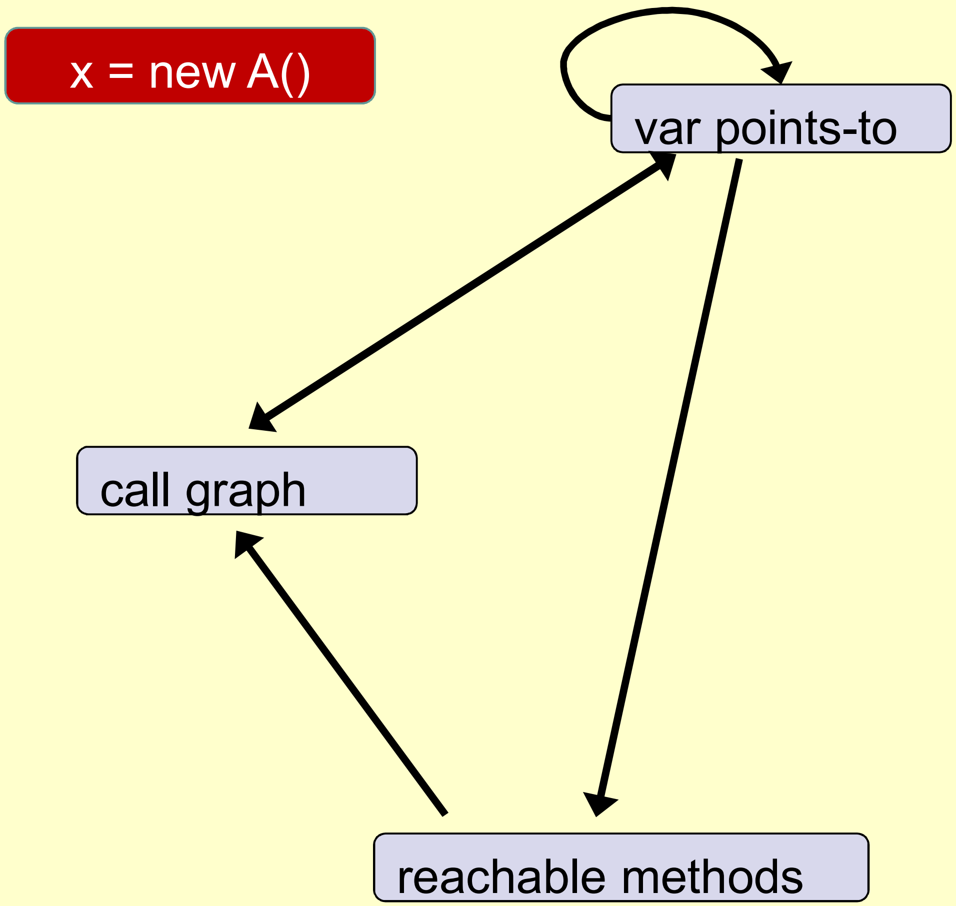
Program Analysis: a Domain of Mutual Recursion



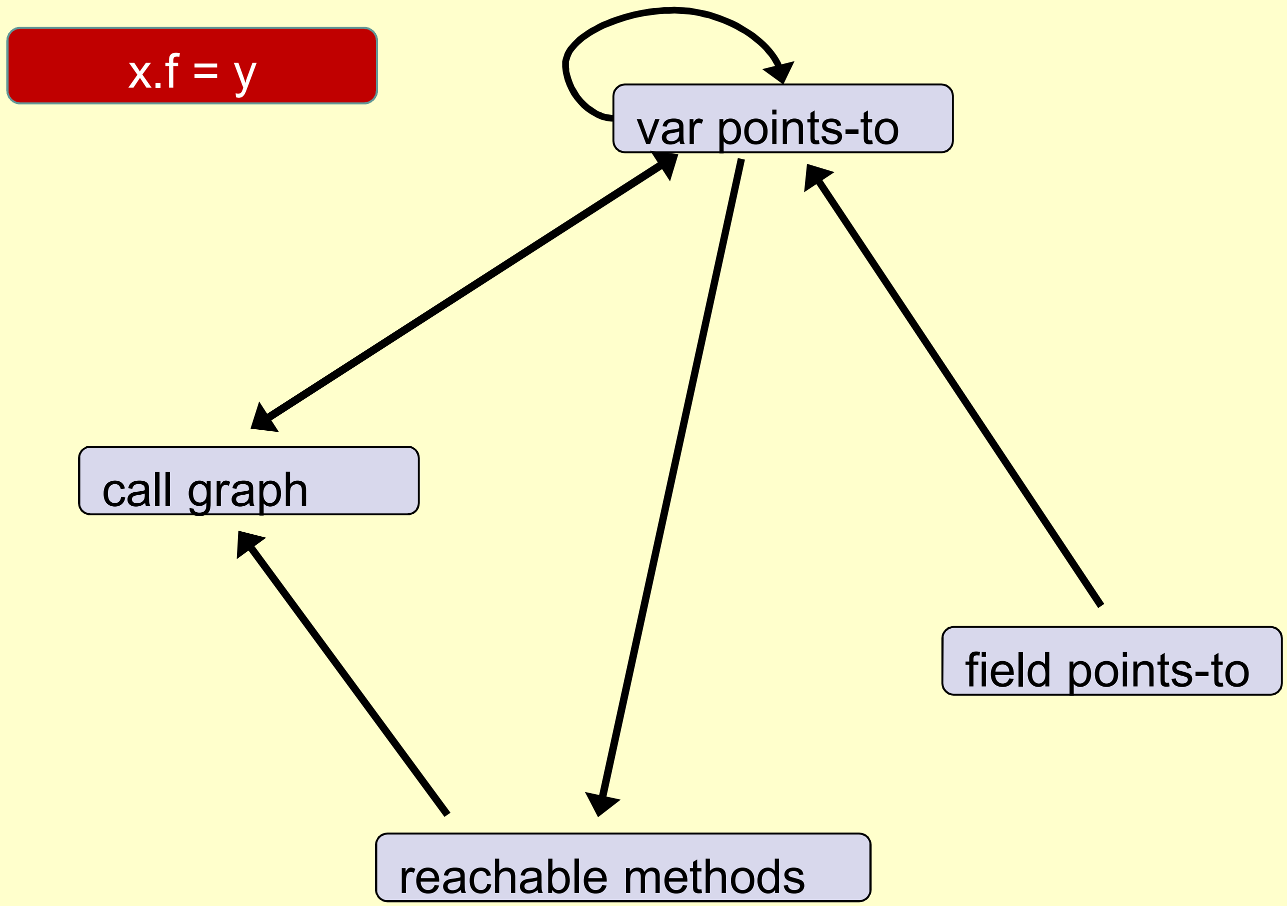
Program Analysis: a Domain of Mutual Recursion



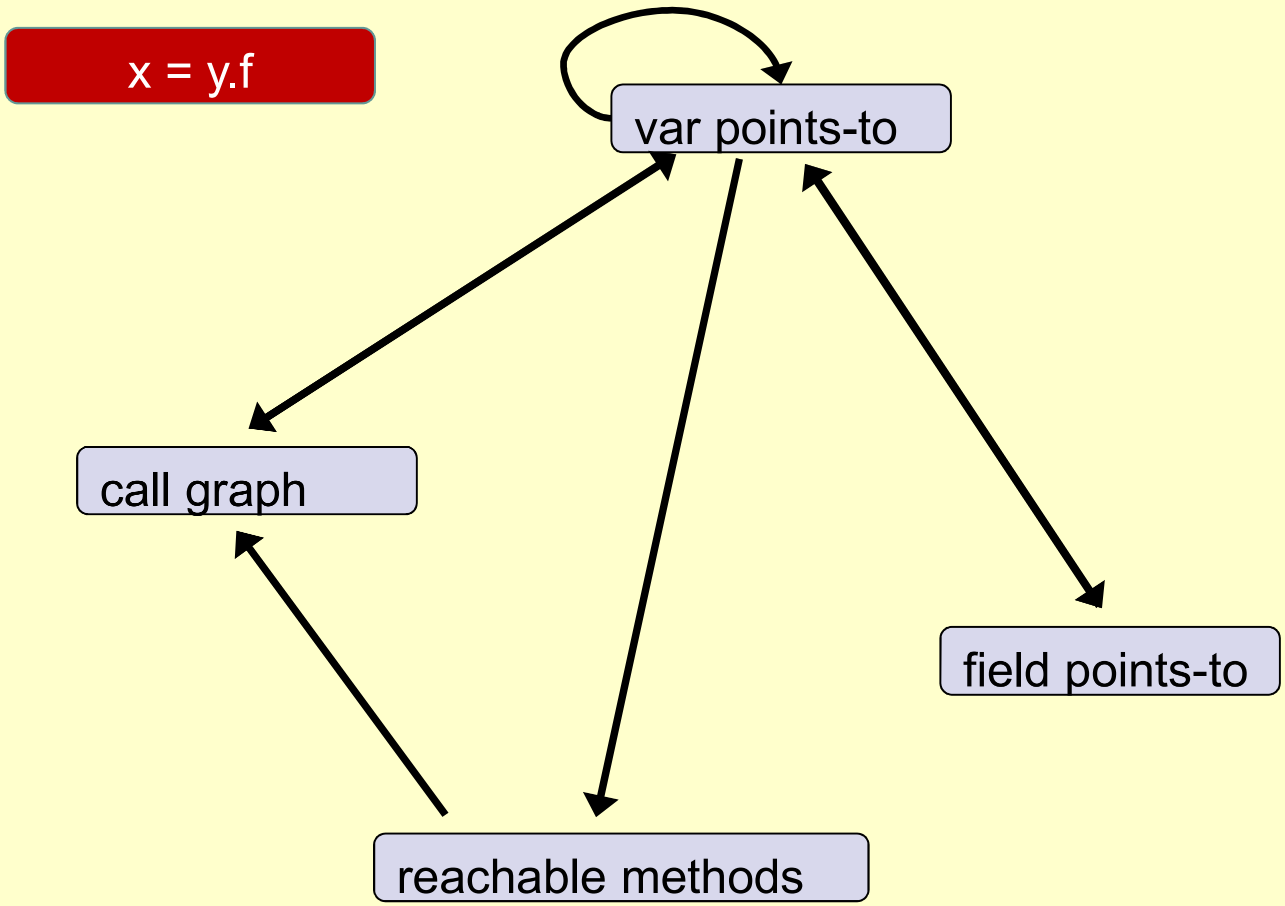
Program Analysis: a Domain of Mutual Recursion



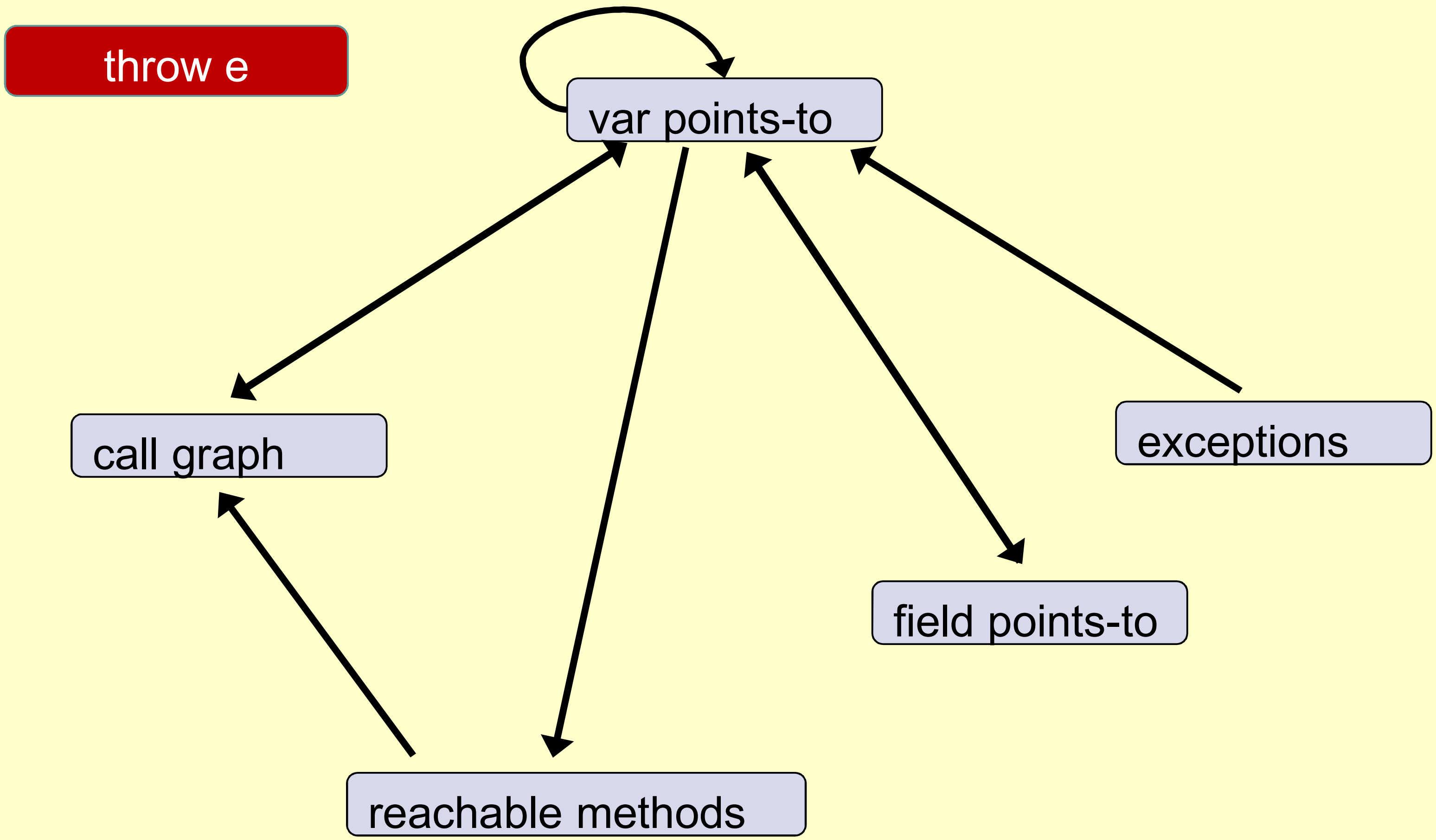
Program Analysis: a Domain of Mutual Recursion



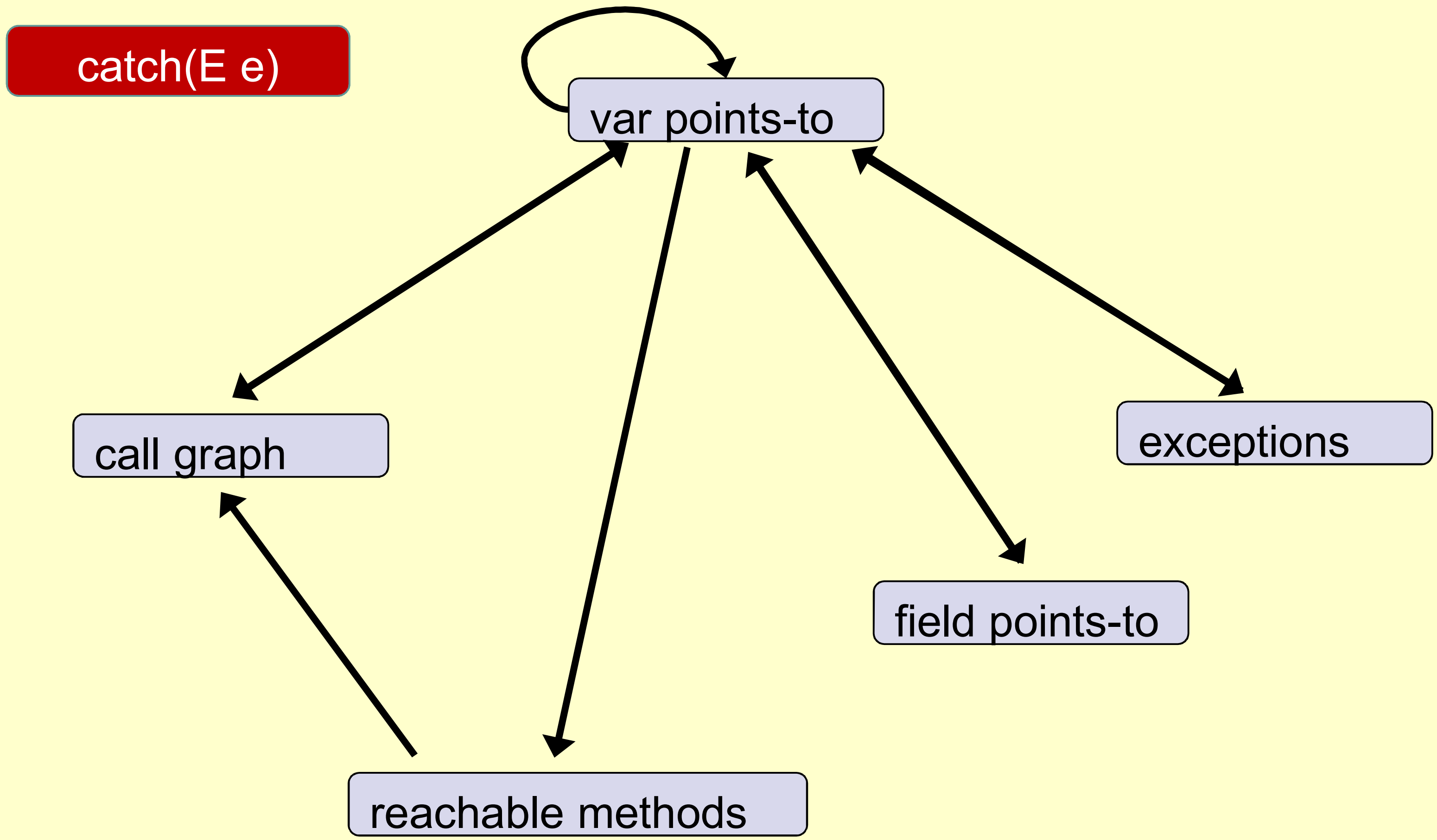
Program Analysis: a Domain of Mutual Recursion



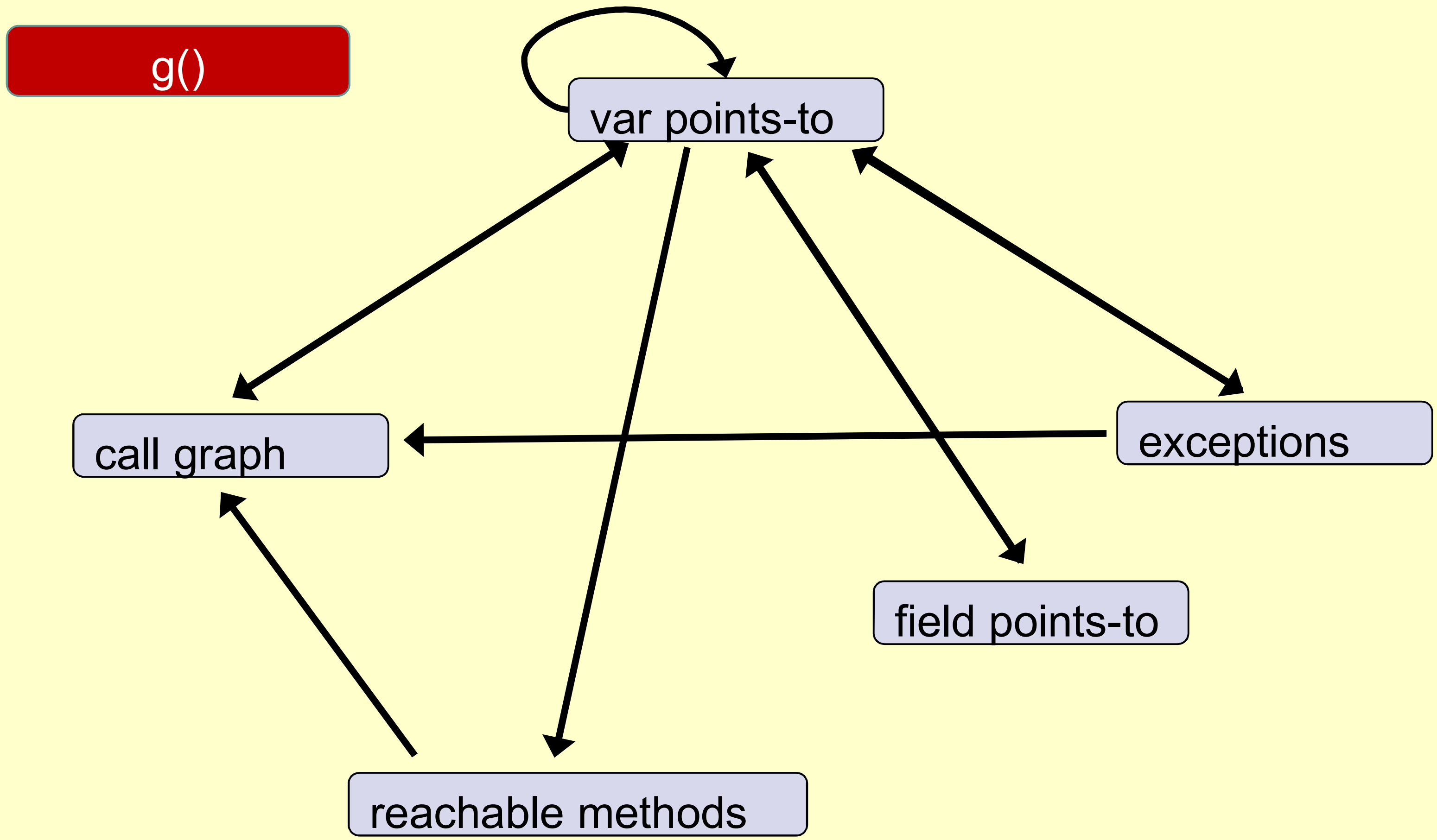
Program Analysis: a Domain of Mutual Recursion



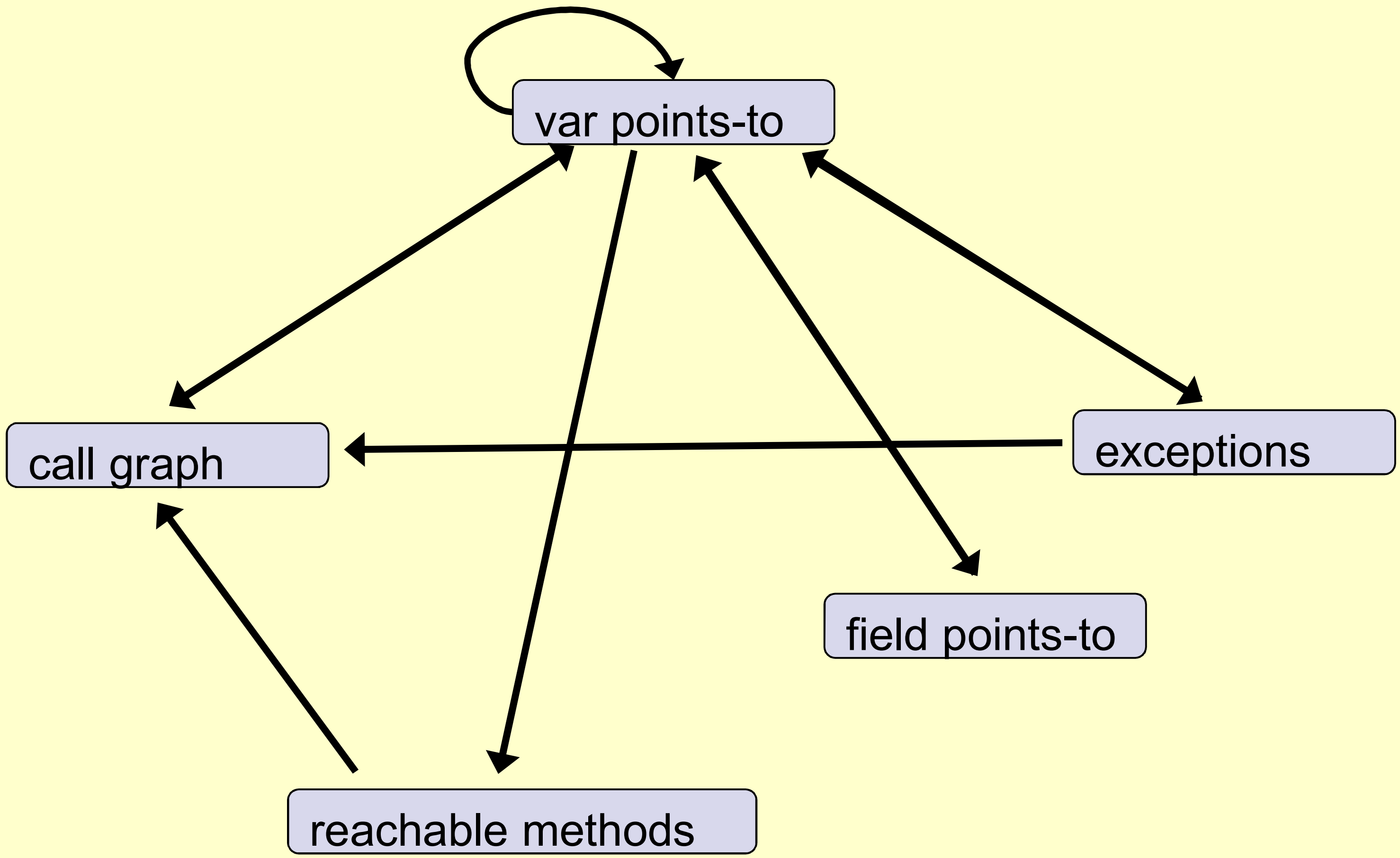
Program Analysis: a Domain of Mutual Recursion



Program Analysis: a Domain of Mutual Recursion



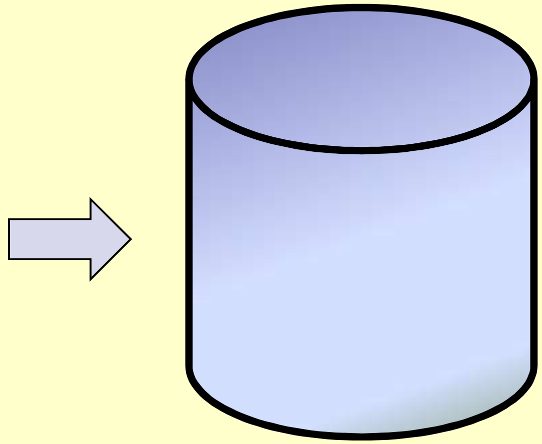
Program Analysis: a Domain of Mutual Recursion



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()

VarPointsTo

Move

a	b
b	a
c	b

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 variables

```
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).
```

Datalog: Properties

- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - increases algorithm space
    - enables different execution strategies, aggressive optimization

Less programming, more specification

The DOOP Framework

- Datalog-based pointer analysis framework for Java
- Declarative: what, not how
- Sophisticated, very rich set of analyses
  - subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive heap, abstraction, type filtering, precise exception analysis
- Support for full semantic complexity of Java
  - jvm initialization, reflection analysis, threads, reference queues, native methods, class initialization, finalization, cast checking, assignment compatibility

http://doop.program-analysis.org

Some Doop Contributions

- Expressed complete, complex pointer analyses in Datalog
  - core specification: ~1500 logic rules
  - parameterized by a handful of rules per analysis flavor
- Synthesized efficient algorithms from specification
  - order of magnitude performance improvement
  - allowed to explore more analyses than past literature
- Approach: heuristics for searching algorithm space
  - targeted at recursive problem domains
- Demonstrated scalability with explicit representation
  - Contrast: previous work used BDDs (bddbddb, Paddle)

Not Expected

- Expressed complete, complex pointer analyses in Datalog

  “*[E]ncoding all the details of a complicated program analysis problem [on-the-fly call graph construction, handling of Java features] purely in terms of subset constraints may be difficult or impossible.” (Lhotak)*

- Scalability and Efficiency

  “Efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in data structures and algorithms”

How Datalog Executes*

***Note: we will focus on bottom-up execution, as used in Doop. A top-down execution strategy is possible, and is useful for other applications of Datalog**

Naïve Datalog Execution

- Execute each rule against all facts in the current database
- Add the resulting facts to the database
- Repeat until no more facts are added

- Efficiency problem
  - Facts inferred in the first iteration are re-inferred in all subsequent iterations
  - In second iteration, need only consider rule instantiations based on at least one new fact

Semi-Naïve Datalog Execution

- Execute program incrementally
  - For each rule r, for each fact f produced in relation R in the previous round, generate all facts inferable via rule r, fact f, and all current facts for relations other than R
- Example
  - Original program

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

  - Semi-Naïve rewriting

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

  - Note that we don't have a similar Δ rule for Move, because Move represents program text and we never get more facts there

Semi-Naïve Datalog Execution

- Rewritten program

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

- Initial facts

```
AssignHeapAllocation(“y”, “o1”).
AssignHeapAllocation(“z”, “o2”).

Move(“x”, “y”).
Move(“y”, “z”).
```

Practice: Semi-Naïve Execution

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

VarPointsTo(to, obj) <-
  LoadField(base, field, to),
  VarPointsTo(base, baseobj),
  FieldPointsTo(baseobj, field, obj).

FieldPointsTo(baseobj, field, obj) <-
  StoreField(base, field, from),
  VarPointsTo(base, baseobj),
  VarPointsTo(from, obj).
```

```
AssignHeapAllocation("x", "o1").
AssignHeapAllocation("y", "o2").

LoadField("x", "f", "x").  % x.f = x
StoreField("x", "f", "y"). % y = x.f
```

Practice: Writing Reachability Analysis

- Reachable(meth) % method implementation meth is reachable
- VarPointsTo(var, obj) % var may point to an object allocated at obj
- Alloc(var, obj, meth) % obj: var = new ...(...) in meth
- VCall(base, sig, inMeth)% virtual call: base.sig(...) in "inMeth"
- Lookup(obj, sig, meth) % looks up the method implementation for allocation
% site obj and method signature sig

- var, base - variable
- obj - object allocation site
- meth, inMeth - method implementation
- sig - method name and signature

- Give rules for Reachable and VarPointsTo, assuming Alloc, VCall, and Lookup are given

Encoding Context-Sensitive Pointer Analysis in Datalog

[PLDI’10, POPL’11, CC’13, PLDI’13, PLDI’14]

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

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)

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

obj: var = new Something();

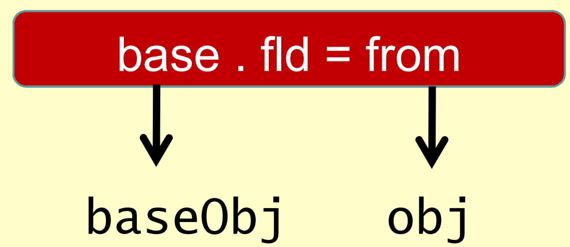
Rule 2: Variable Assignment
(Move)

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

to = from

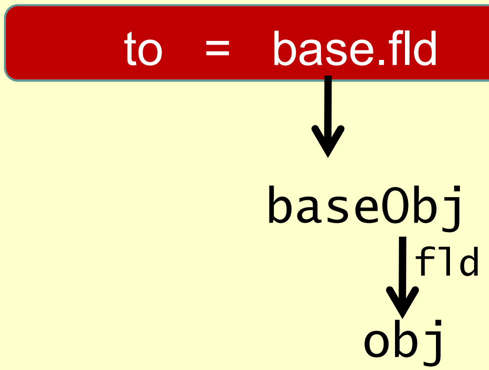
Rule 3: Object Field Write
(Store)

```
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).
```



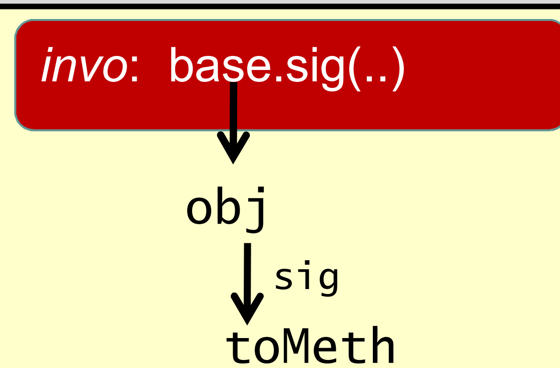
Rule 5: Static Method Calls (`SCall`)

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

invo: toMeth(..)

Rule 6: Virtual Method Calls
(VCall)

```
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).
where calleeCtx = Merge(obj, hctx, invo, callerCtx)
```



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:
  - $\mathit{Context} = \mathit{HContext} = \mathrm{Instr}$
- 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:
  - $Context = HContext = \mathrm{Instr}$
- 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:
  - $Context = \mathrm{Instr}^2$, $HContext = \mathrm{Instr}$
- Functions:
  - ***Record***`(obj, ctx) = first(ctx)`
  - ***Merge***`(obj, hctx, invo, callerCtx) =`
    `pair(obj, first(callerCtx))`
  - ***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*)

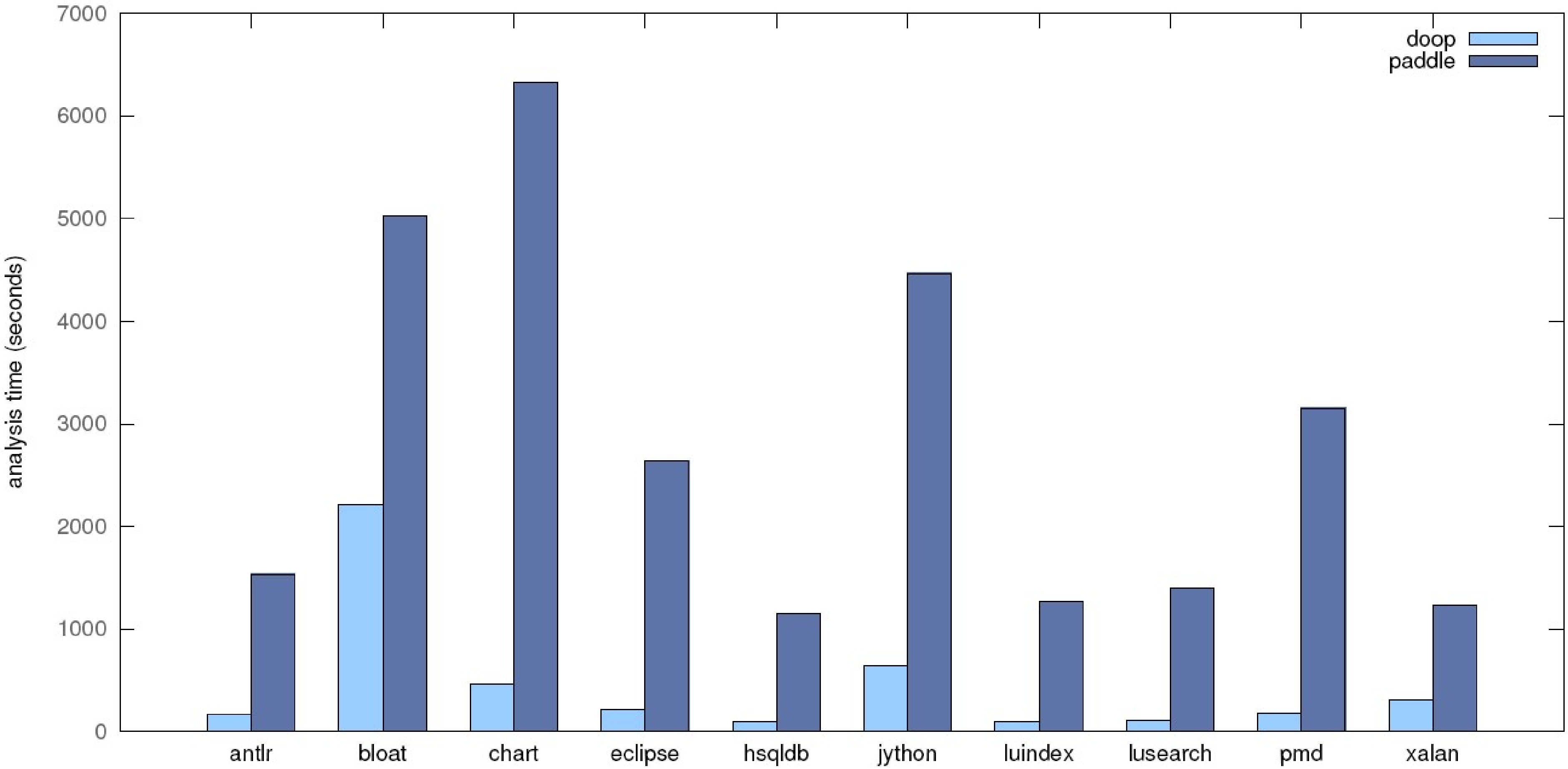
Impressive Performance, Implementation Insights

[OOPSLA’09, ISSTA’09]

Impressive Performance

- Compared to Paddle
  - most complete, scalable past framework
    - includes analyses with a context sensitive heap
- Large speedup for fully equivalent results
  - 15.2x faster for 1-obj, 16.3x faster for 1-call, 7.3x faster for 1-call+heap, 6.6x faster for 1-obj+heap
- Large speedup for more precise results!
  - 9.7x for 1-call, 12.3x for 1-call+heap, 3x for 1-obj+heap
- Scaling to analyses Paddle cannot handle
  - 2-call+1-heap, 2-object+1-heap, 2-call+2-heap

1-call-site-sensitive+heap



Where Is The Magic?

- Surprisingly, in very few places
  - 4 orders of magnitude via optimization methodology for highly recursive Datalog!
    - straightforward data processing optimization (indexes), but with an understanding of how Datalog does recursive evaluation
  - no BDDs
    - are they needed for pointer analysis?
  - simple domain-specific enhancements that increase both precision and performance in a direct (non-BDD) implementation

Identifying Performance Problems

- Inefficient Datalog:

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

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

  - LogicBlox Datalog indexes on the *last* variable in a relation
  - This code joins on the *first* variables of Assign, VarPointsTo

- After re-ordering variables:

```
VarPointsTo(?obj, ?var) <-
  AssignObjectAllocation(?obj, ?var).

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

More Challenging

- Specification

```
FieldPointsTo(?obj, ?field, ?baseobj) <-
  StoreField(?from, ?field, ?base),
  VarPointsTo(?baseobj, ?base),
  VarPointsTo(?obj, ?from).
```

- Semi-naïve executions

```
ΔFieldPointsTo(?obj, ?field, ?baseobj) <-
  StoreField(?from, ?field, ?base),
  ΔVarPointsTo(?baseobj, ?base),
  VarPointsTo(?obj, ?from).

ΔFieldPointsTo(?obj, ?field, ?baseobj) <-
  StoreField(?from, ?field, ?base),
  VarPointsTo(?baseobj, ?base),
  ΔVarPointsTo(?obj, ?from).
```

Folding

- Original

```
FieldPointsTo(?obj, ?field, ?baseobj) <-
  StoreField(?from, ?field, ?base),
  VarPointsTo(?baseobj, ?base),
  VarPointsTo(?obj, ?from).
```

- Implemented with an extra relation
  - and an extra index

```
FieldPointsTo(?obj, ?field, ?baseobj) <-
  StoreObjectField(?baseobj, ?field, ?from)
  VarPointsTo(?obj, ?from).

StoreObjectField(?baseobj, ?field, ?from)
  StoreField(?from, ?field, ?base),
  VarPointsTo(?baseobj, ?base),
```

Optimization Practice

- Original

```
VarPointsTo(?obj, ?to) <-
  LoadField(?to, ?base, ?field),
  VarPointsTo(?baseobj, ?base),
  FieldPointsTo(?obj, ?field, ?baseobj).
```

- Optimize the variable order of LoadField and add folds
  - You may not change variable order in other relations

Optimization Insight

- For highly recursive Datalog programs, ***relation deltas produced by semi-naive evaluation should bind all the variables needed to index into other relations***
- Can require complex reasoning
  - whole program optimization
  - human needs to be in the loop in the search of algorithm space!

Algorithmic Enhancements

- BDDs are necessary if one is not careful about precision
- We introduced simple algorithmic enhancements to avoid redundancy
  - static initializers handled context-insensitively
  - on-the-fly exception handling
- Better analyses, as well as faster!

Conclusions, Future Work

Declarative Program Analysis

- Doop is probably the most complete points-to analysis framework for Java
  - aids exploration
  - competitive performance
- Already yielded new algorithms, implementation techniques
- Latest: flow-sensitive, LLVM, client analyses
- Future: commercialization as program comprehension service