Declarative Static Program Analysis: An Intelligent System over Programs

Yannis Smaragdakis
University of Athens
with
Martin Bravenboer,
George Kastrinis, George Balatsouras,
Tony Antoniadis, George Fourtounis, Neville Grech
and
Kostas Ferles, Nikos Filippakis,
Sifis Lagouvardos, Yue Li, Petros Pathoulas,
Kostas Saidis, Tian Tan, Konstantinos Triantafyllou

erc
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
programme for development
EUROPEAN SOCIAL FUND
Co-financed by Greece and the European Union

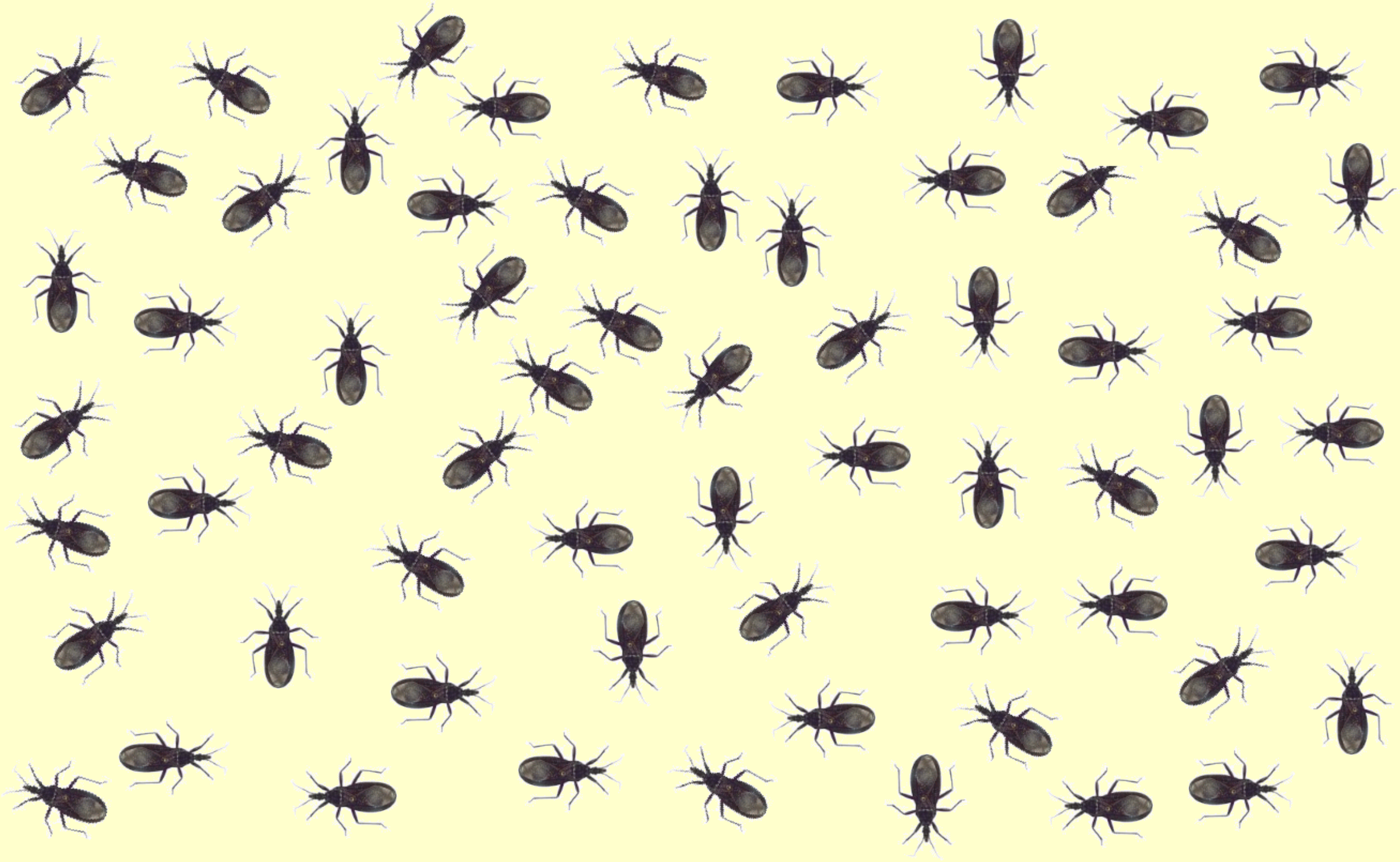
LOGICBLOX

Overview

- What do we do?
  - static program analysis
    - “discover program properties that hold for all executions”
- Vision: a system that knows more about your program than you do
- How do we do it?
  - declarative (logic-based specification)
    - fast, powerful, new insights

Program Analysis: Run Faster
(e.g., compiler optimization)

Program Analysis: Find Bugs



Program Analysis: Software Understanding

(e.g., slicing, refactoring, program queries)

My Research: Doop
and friends: CClyzer, MadMax

- Since 2008:
  - Doop: a powerful framework for analyzing Java bytecode
    - building on *pointer analysis*
      - now just a substrate for more analyses
  - declarative, using the Datalog language
- Lots of offshoots
  - Cclyzer, for LLVM bitcode
  - GigaHorse/MadMax for EVM bytecode

Pointer Analysis
(but really: value-flow analysis)

- What objects can a variable point to?

objects represented by allocation sites

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()

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	ne
id:a	n
foo:b	n
bar:b	n

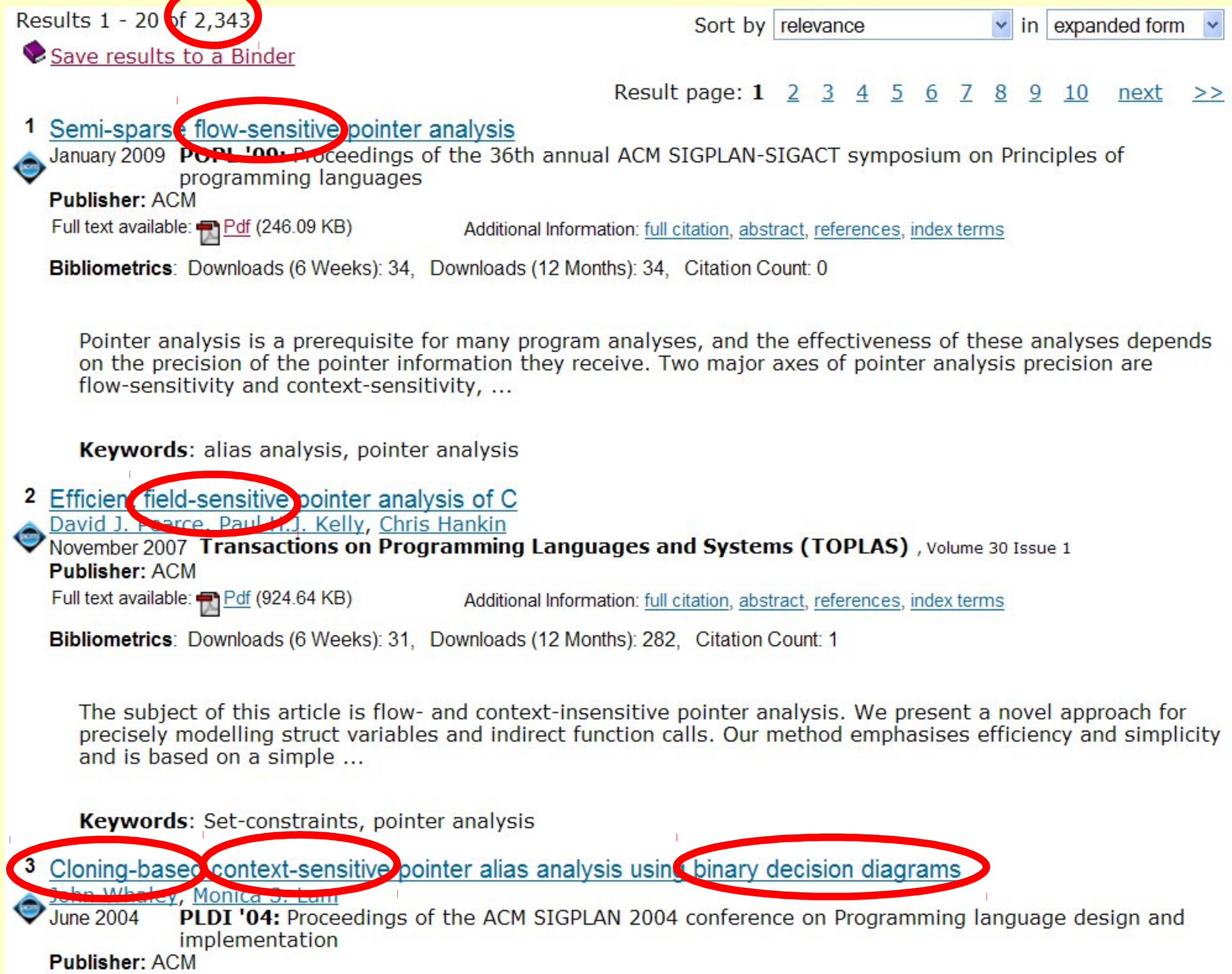
remember for later: context-sensitivity is what makes an analysis precise

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

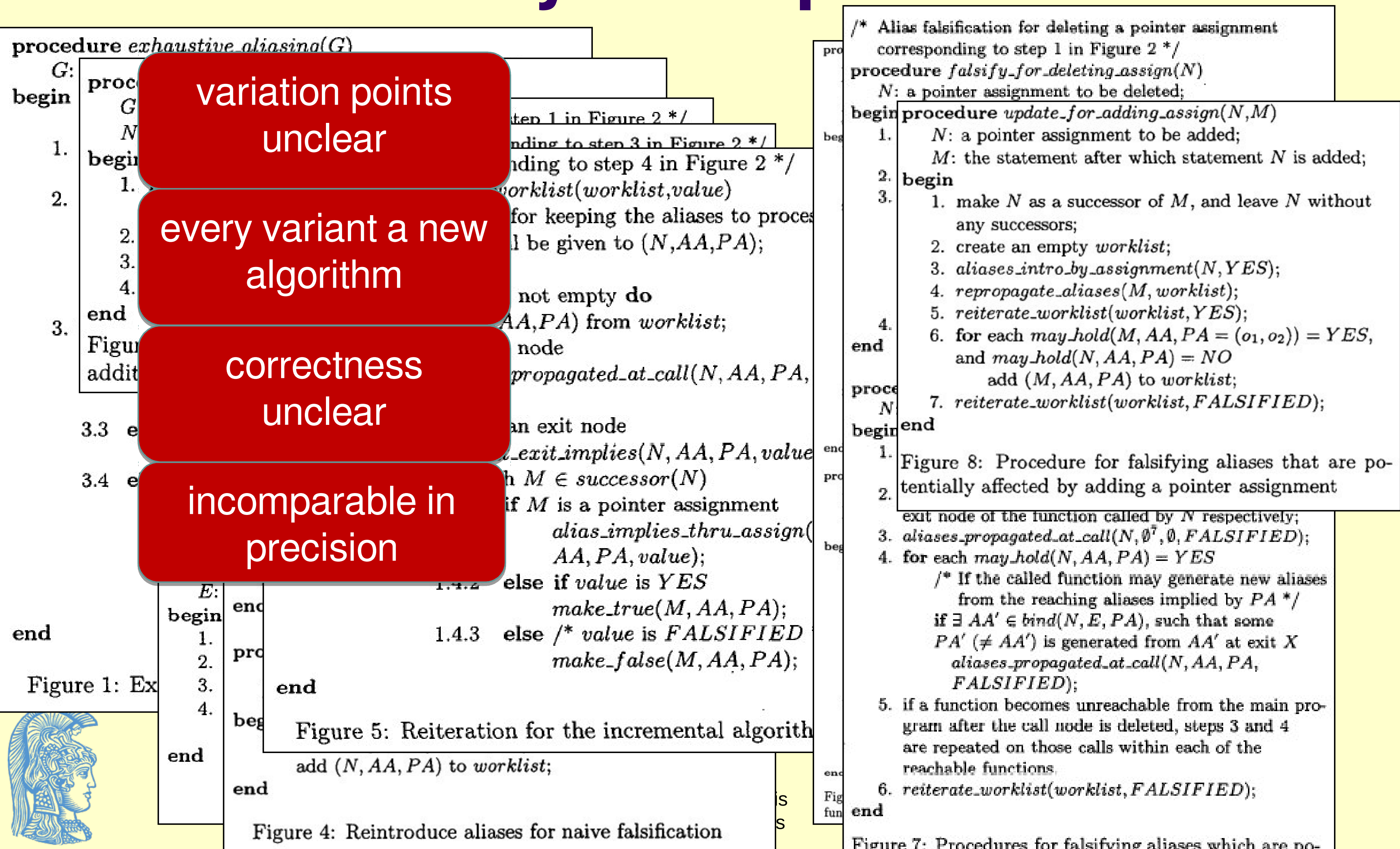
2 Efficient field-sensitive pointer analysis of C
David J. Pearce, Paul H.J. Kelly, Chris Hankin
November 2007 **Transactions on Programming Languages and Systems (TOPLAS)** , Volume 30 Issue 1
Publisher: ACM
Full text available: Pdf (924.64 KB) Additional Information: full citation, abstract, references,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

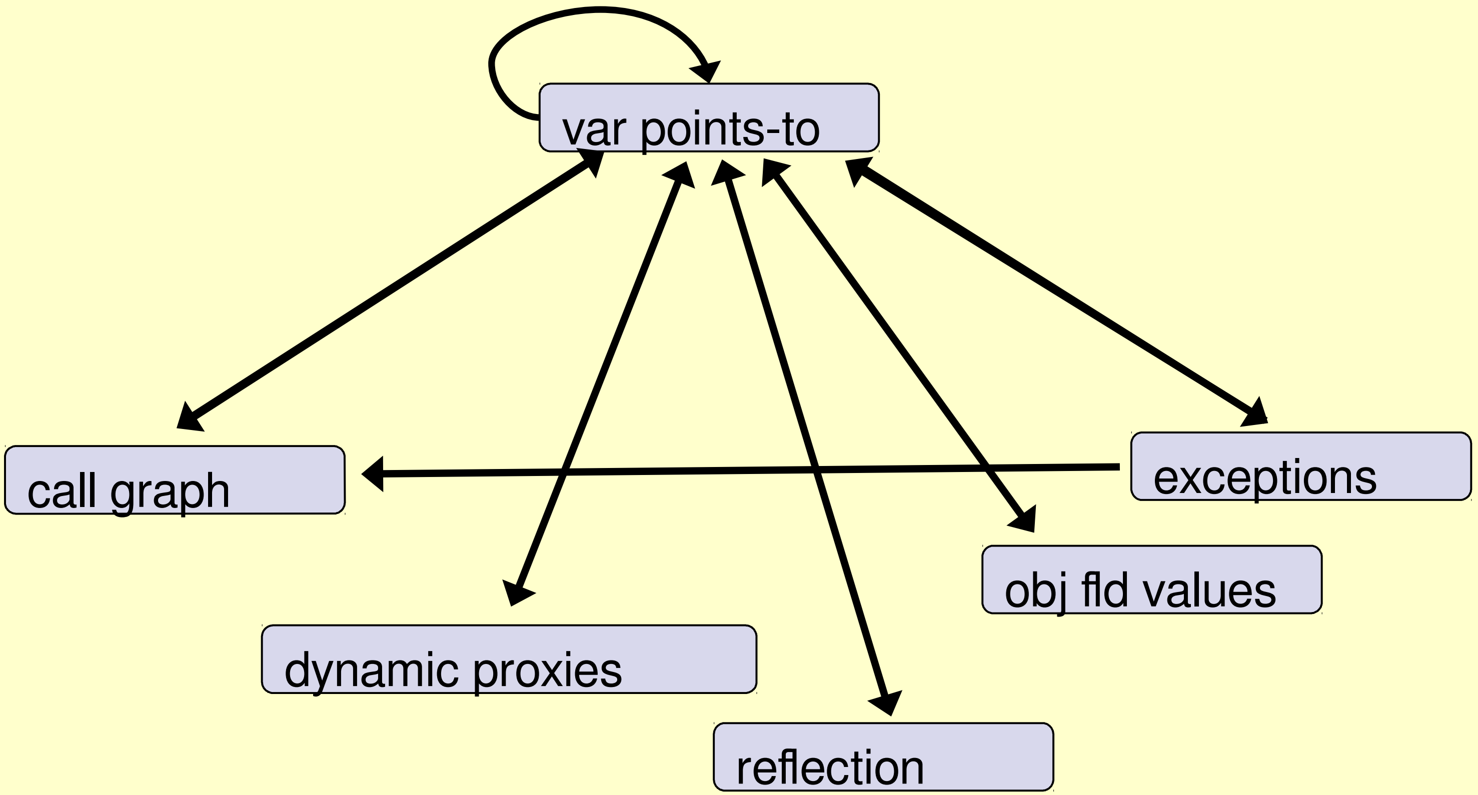
Keywords: Set-constraints, pointer analysis

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



***Holistic Program Analysis*:** "Everything Is Connected"

A Vision Within Reach

- *An intelligent system that knows more about your program than you do*
- “Everything is connected”
  - all analysis aspects encoded separately, all benefitting each other
- The Doop framework serves to illustrate
- Key: a declarative specification of all sorts of static analyses
- In Doop: use of Datalog

Datalog To The Rescue!

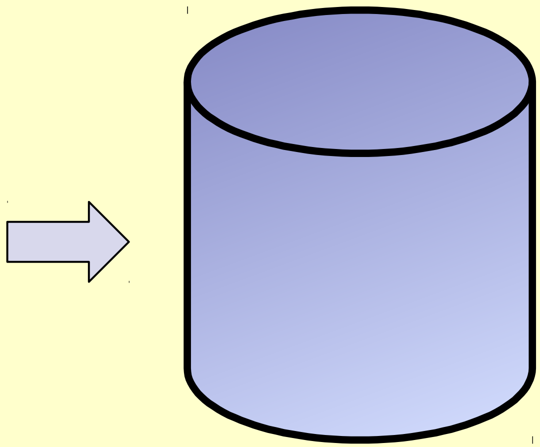
- Datalog is relations + recursion
- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - e.g., as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - monotonic

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()

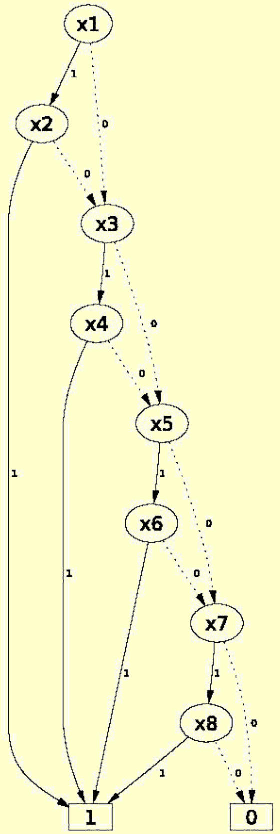
```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

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

The Doop Framework

- Datalog-based static 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

Pointer Analysis: Previous Approaches



Context-sensitive pointer analysis for Java

- paddle
  - Java + relational algebra + binary decision diagrams (BDD)
- WALA
  - Java, conventional approach
- bddbddb (pioneered Datalog for realistic points to analysis)
  - Datalog + Java + BDD

Past Approaches and Declarative Analysis

- Past approaches have flirted with declarative analysis
- But no purely declarative approach
  - specification and algorithm confused
- Declarativeness considered unscalable in both complexity and performance
  - "*the first time I write an analysis it is typically in Datalog, but then, once I'm convinced it's precise, I throw it out and I write it in Java, when I want to focus on scalability.*" (*Naik, 2010*)

Doop Makes Declarative Analysis Real

- Complete, complex pointer analyses in Datalog
  - core specification: ~1500 logic rules
  - parameterized by a handful of rules per analysis flavor
- 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
  - no BDDs

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"

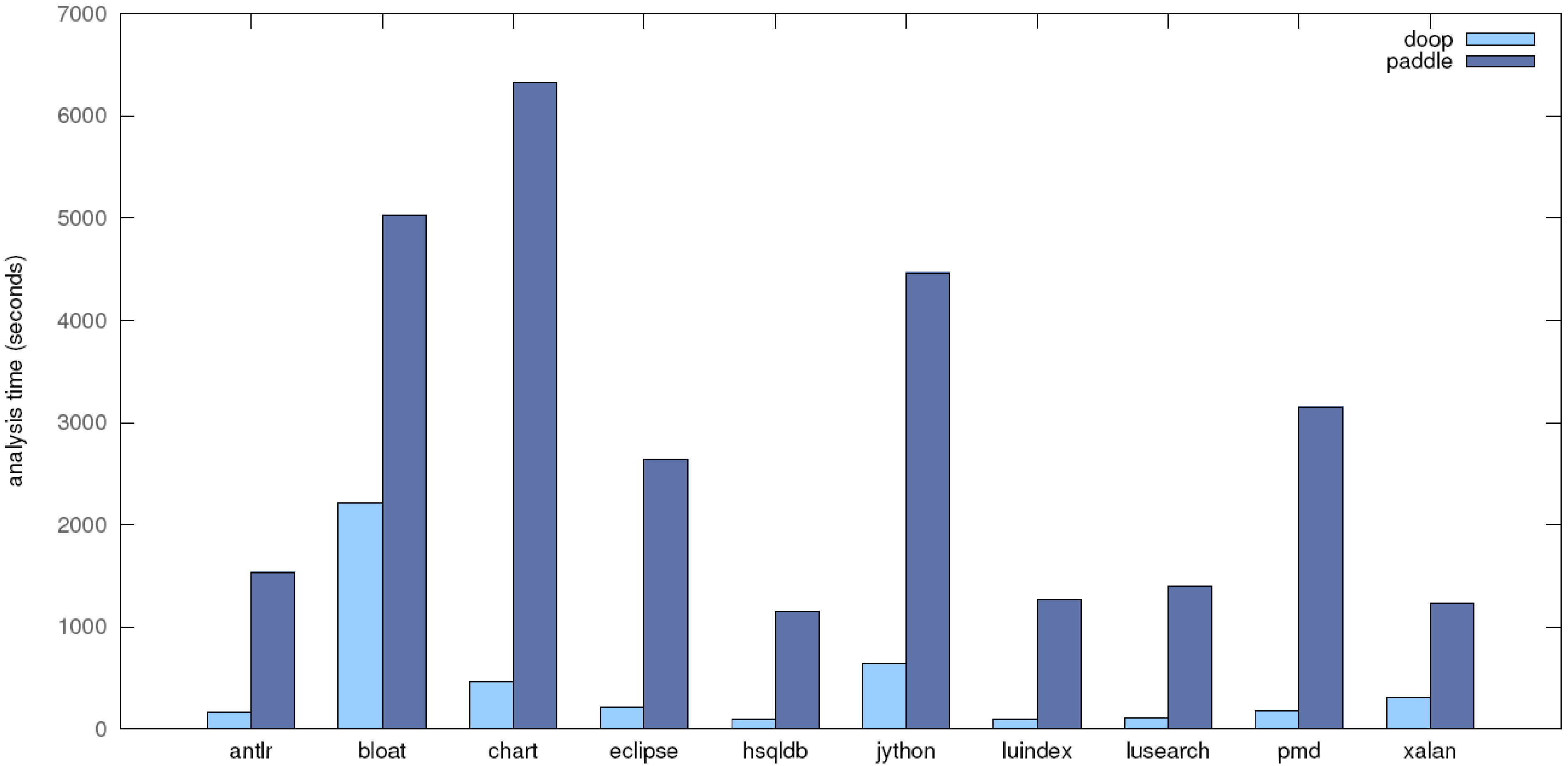
Flyover Tour of Interesting Results

What have we done with this?

Impressive Performance, Implementation Insights

[OOPSLA'09, ISSTA'09]

Large Speedup For Realistic Analyses



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

Better Understanding of Existing Algorithms, More Precise and Scalable New Algorithms

[PLDI’10, POPL’11, CC’13, PLDI’13, PLDI’14, FSE'18, OOPSLA'18]

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

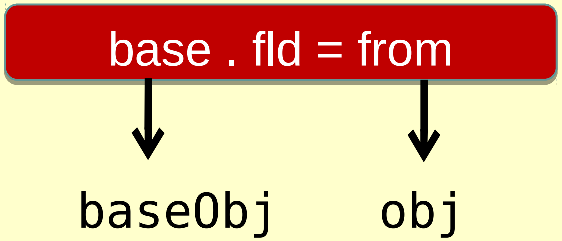
(Move)

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

to = from

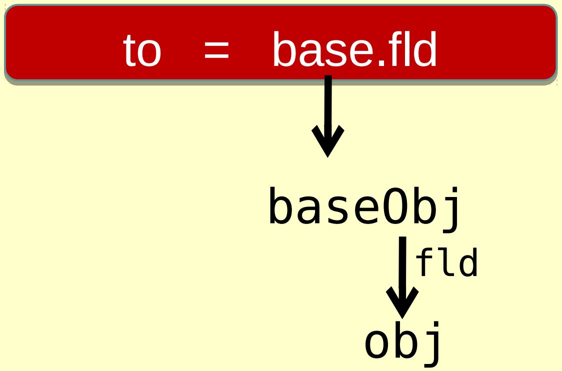
Rule 3: Object Field Write (Store)

```
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).
```



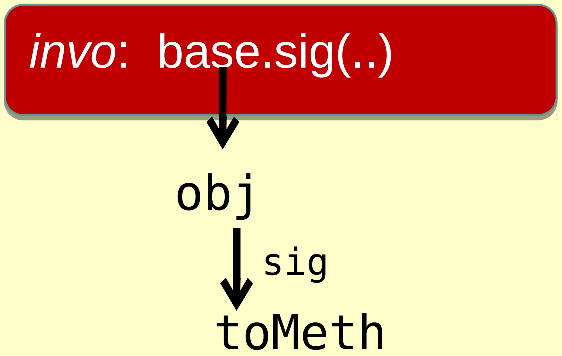
Rule 5: Static Method Calls
(SCall)

```
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*)

Many More Work Threads

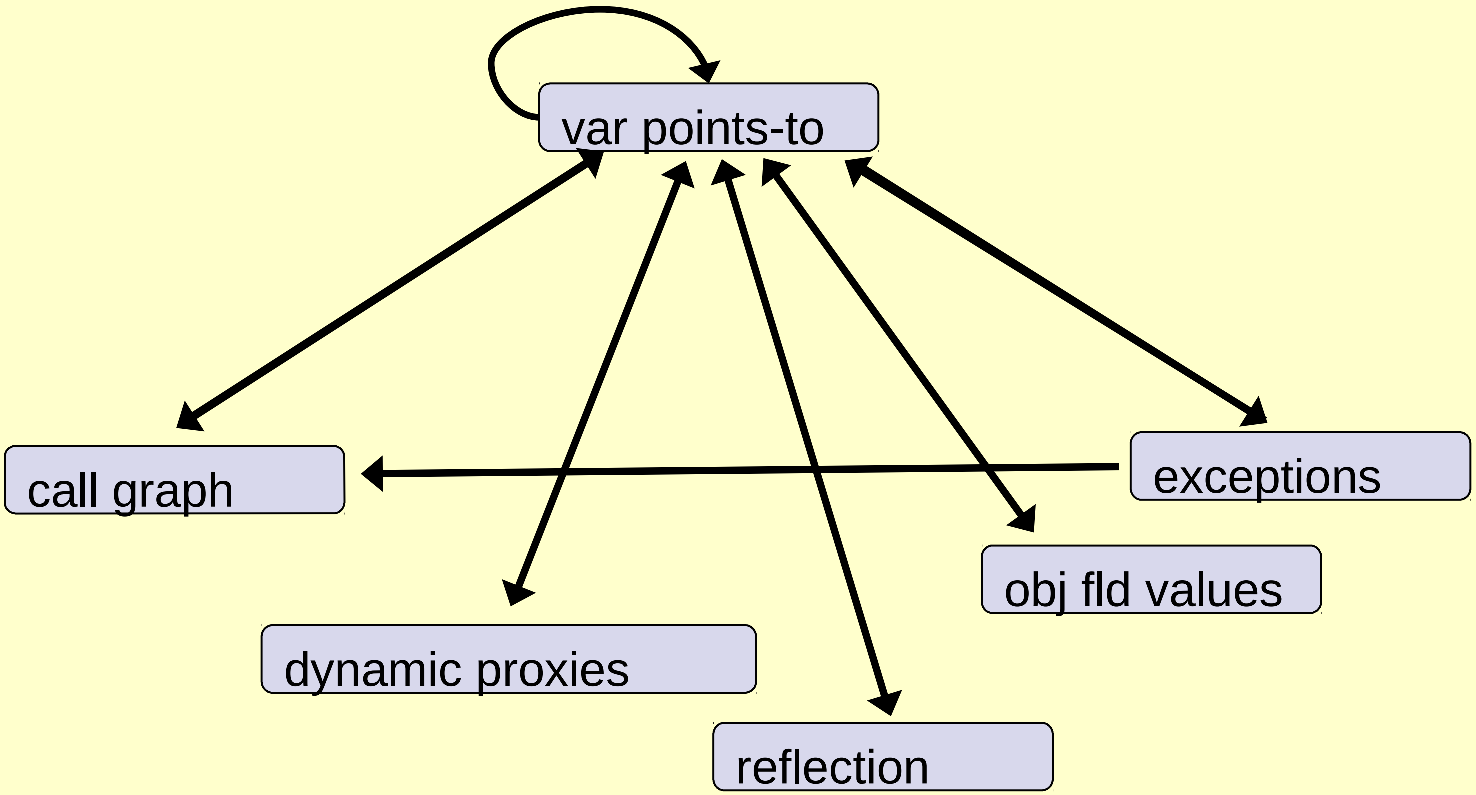
- Set-based pre-analysis [OOPSLA’13]
  - universal optimization technique
- Completing a partial program [OOPSLA’13]
  - making sense out of missing libraries
- Soundness [CACM 2/15, ECOOP'18 **(distinguished paper)**]
- Reflection and dynamic loading [APLAS'15, ECOOP'18, ISSTA'18]
- Port to Souffle: a parallel Datalog engine [SOAP’17]
- Must-alias analysis [SOAP’17, CC'18]
- Taint analysis using points-to algorithms! [OOPSLA’17]
- Integrating heap snapshots in static analysis [OOPSLA’17, ISSTA'18]

Summary and Vision

The Vision

- Doop: early instance of intelligent system that just *knows* things about your program
- The use of Datalog fits very well
  - knowledge-base
  - a database of inferences
  - rules that (if correct) apply independently of others
    - yet in mutual recursion
    - monotonically

Mutual Recursion: Cannot Emphasize Enough



Holistic Program Analysis: "Everything Is Connected"