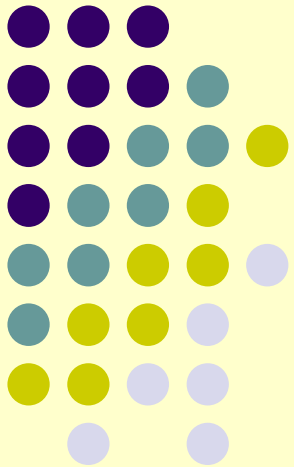
Declarative Static Program Analysis

Yannis Smaragdakis
University of Athens

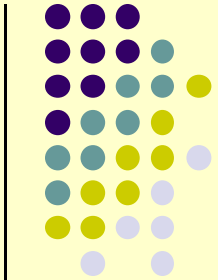
joint work with Martin Bravenboer,
George Kastrinis,
George Balatsouras



European Research Council
Established by the European Commission

Co-financed by Greece and the European Union

Overview



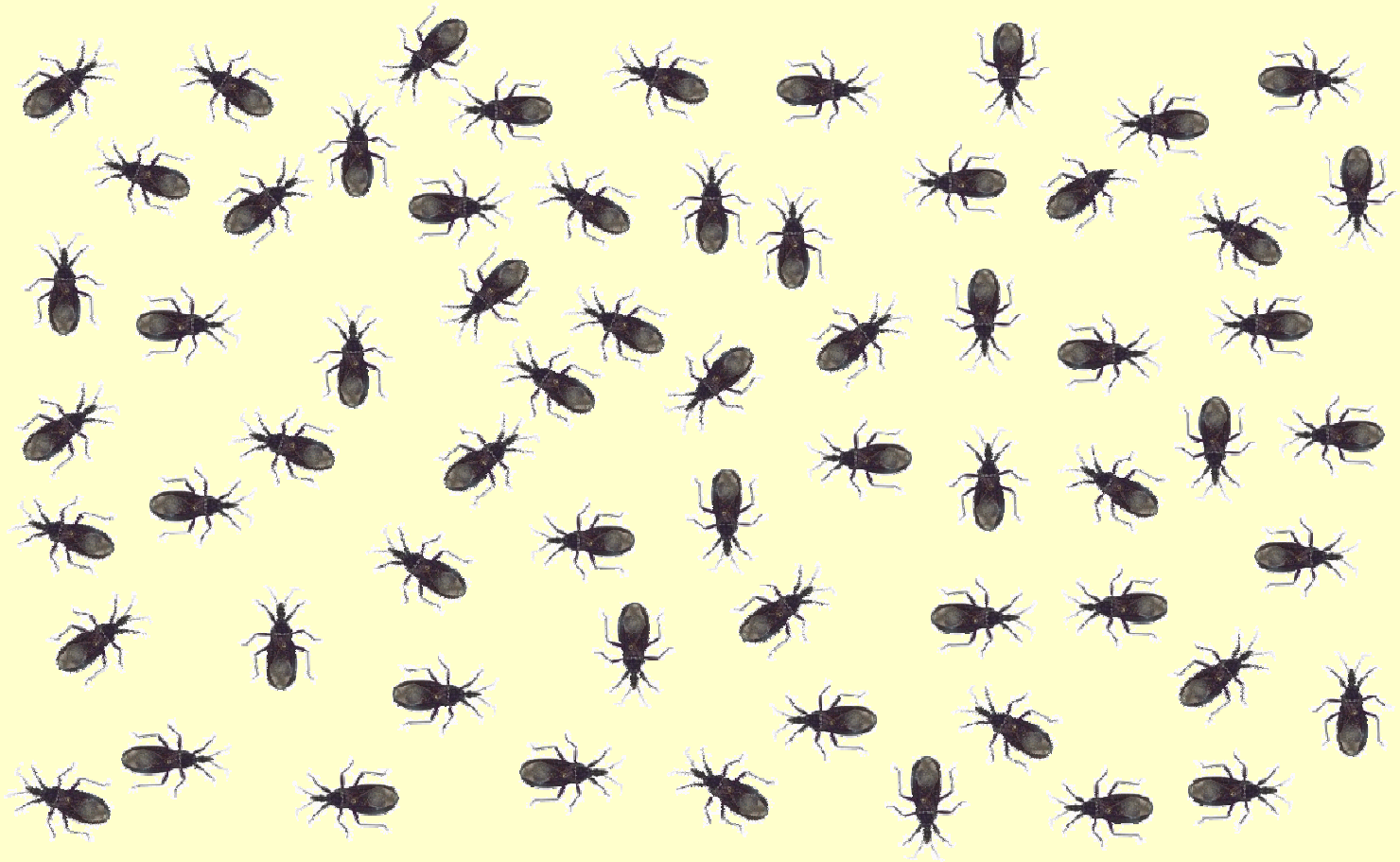
- What do we do?
  - static program analysis
    - “discover program properties that hold for all executions”
  - declarative (logic-based specification)
- Why do you care?
  - simple, very powerful
  - screaming fast!
  - different, major lessons learned
    - several new algorithms, optimization techniques, implementation insights (no BDDs)

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

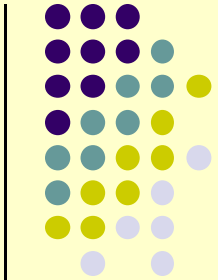
Program Analysis: Software Understanding

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

Program Analysis: Find Bugs



Our Work



- In the past 5 years:
  - Doop: a very powerful framework for Java *pointer analysis*
    - the mother of all sophisticated static analyses
  - declarative, using the Datalog language
  - some work on client analyses
- In the future:
  - analyses for other languages
  - lots of other low- and high-level analyses

Pointer Analysis

- What objects can a variable point to?

objects represented by allocation sites

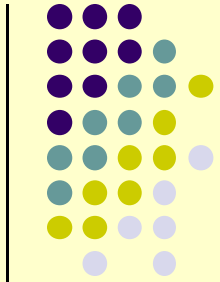
program

```
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?

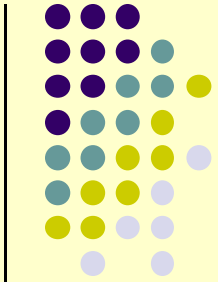
program

```
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	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

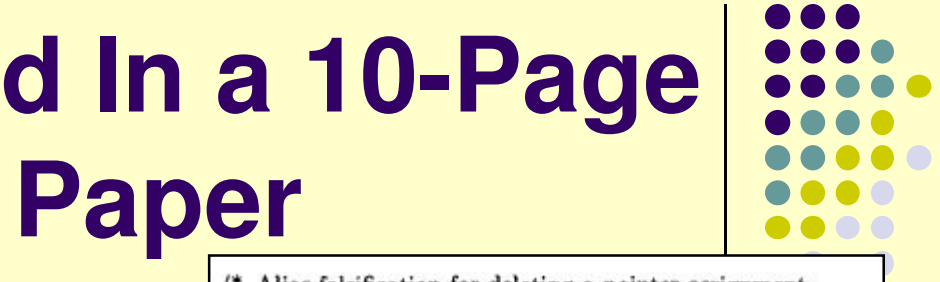
3 Cloning-based context-sensitive pointer alias analysis using binary decision diagrams

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



variation points unclear

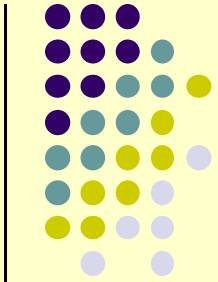
every variant a new algorithm

correctness unclear

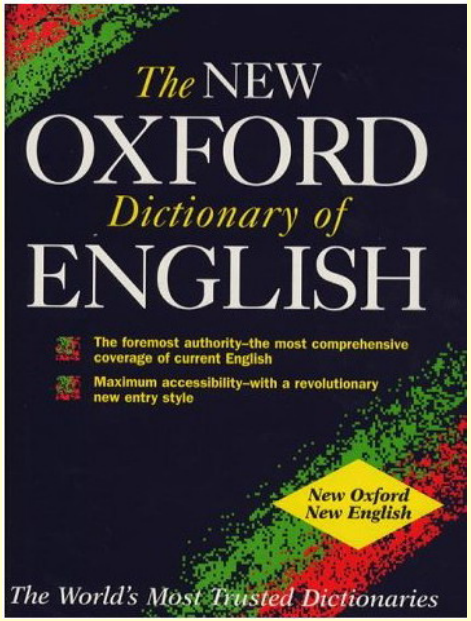
incomparable in precision

Yannis Smaragdakis
University of Athens

What Does It Mean To Be Declarative?



"denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed."

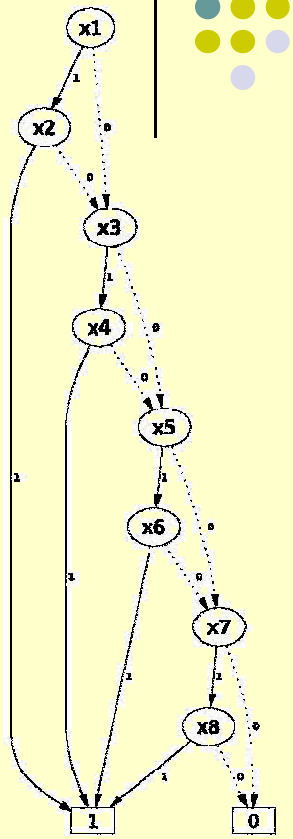


- high-level
- what, not how
- no control-flow
- no side-effects
- specifications, not programs, not algorithms

Pointer Analysis: Previous Approaches

Context-sensitive pointer analysis for Java

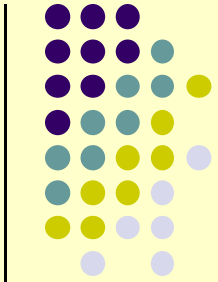
- paddle
  - Java + relational algebra + binary decision diagrams (BDD)
- wala
  - Java, conventional approach
- bddbddb (pioneered Datalog for realistic points to analysis)
  - Datalog + Java + BDD



not a single purely declarative approach

coupling of specification and algorithm

Our Framework



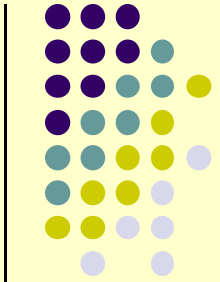
- Datalog-based pointer analysis framework for Java
- Declarative: what, not how

DOOP

- 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

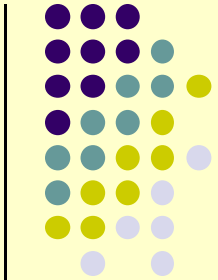
http://doop.program-analysis.org

Key Contributions



- Expressed complete, complex pointer analyses in Datalog
  - core specification: ~600 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
  - no BDDs

These Contributions Are Surprising



- Expressed complete, complex pointer analyses in Datalog

  Lhotak: *"[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."*

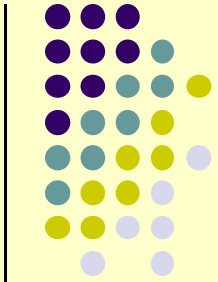
- Scalability and Efficiency

  Lhotak: *"Efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in data structures and algorithms"*

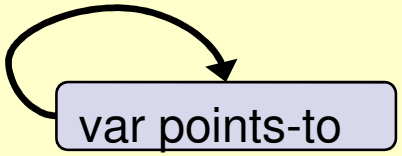
  Whaley: *"Owing to the power of the BDD data structure, bddbddb can even solve analysis problems that were previously intractable"*

  Lhotak: *"I've never managed to get Paddle to run in available memory with these settings [2-cfa context-heap], at least not on real benchmarks complete with the standard library."*

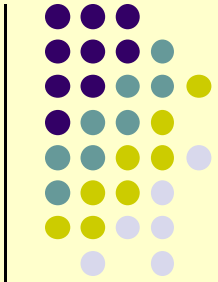
Program Analysis: a Domain of Mutual Recursion

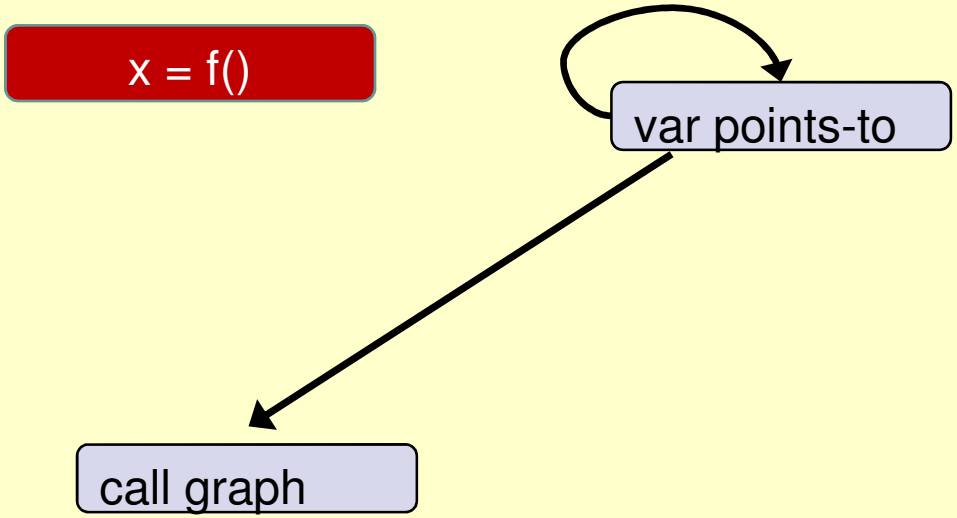


x = y

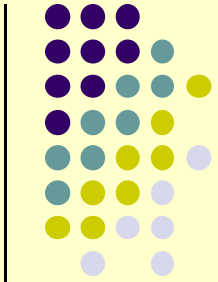


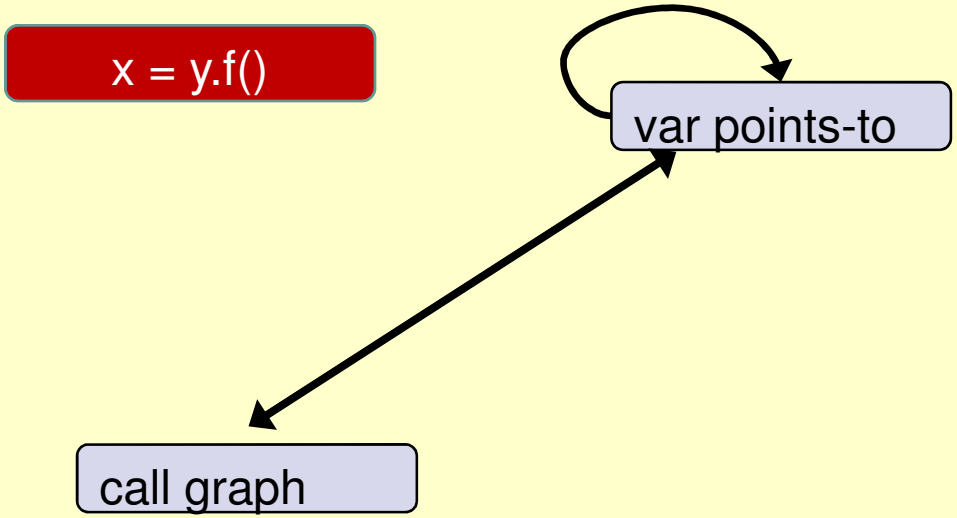
Program Analysis: a Domain of Mutual Recursion



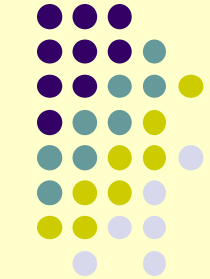


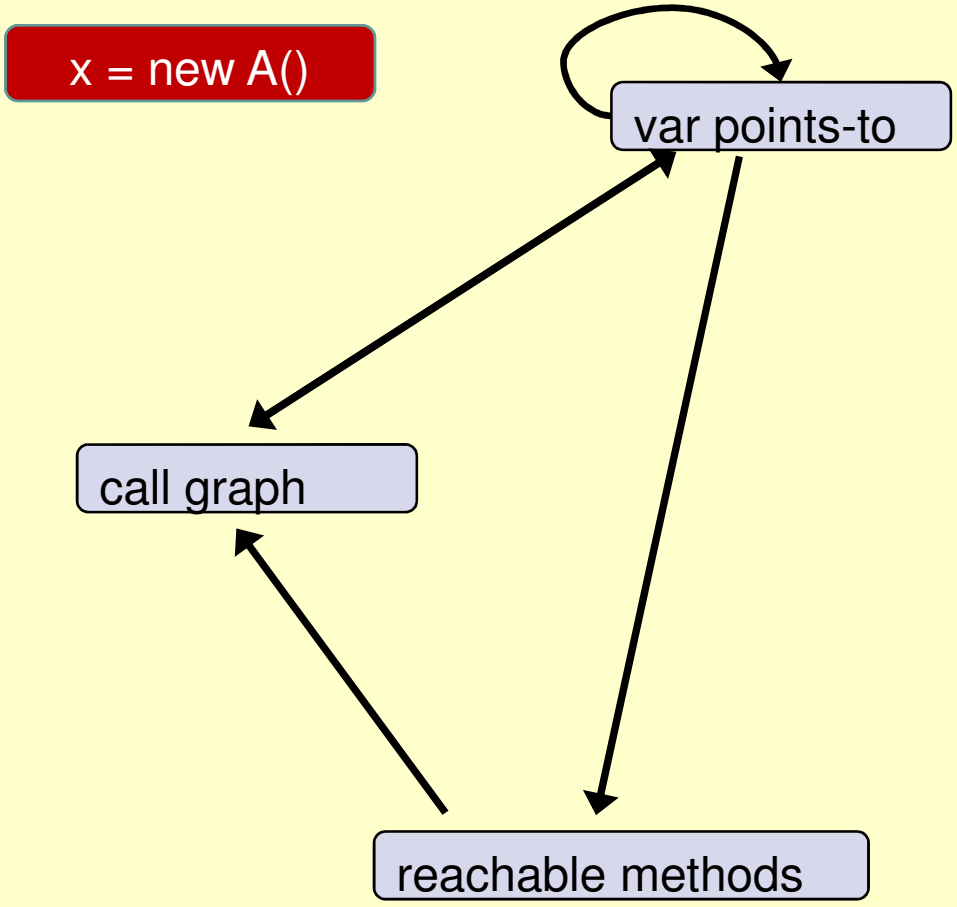
Program Analysis: a Domain of Mutual Recursion



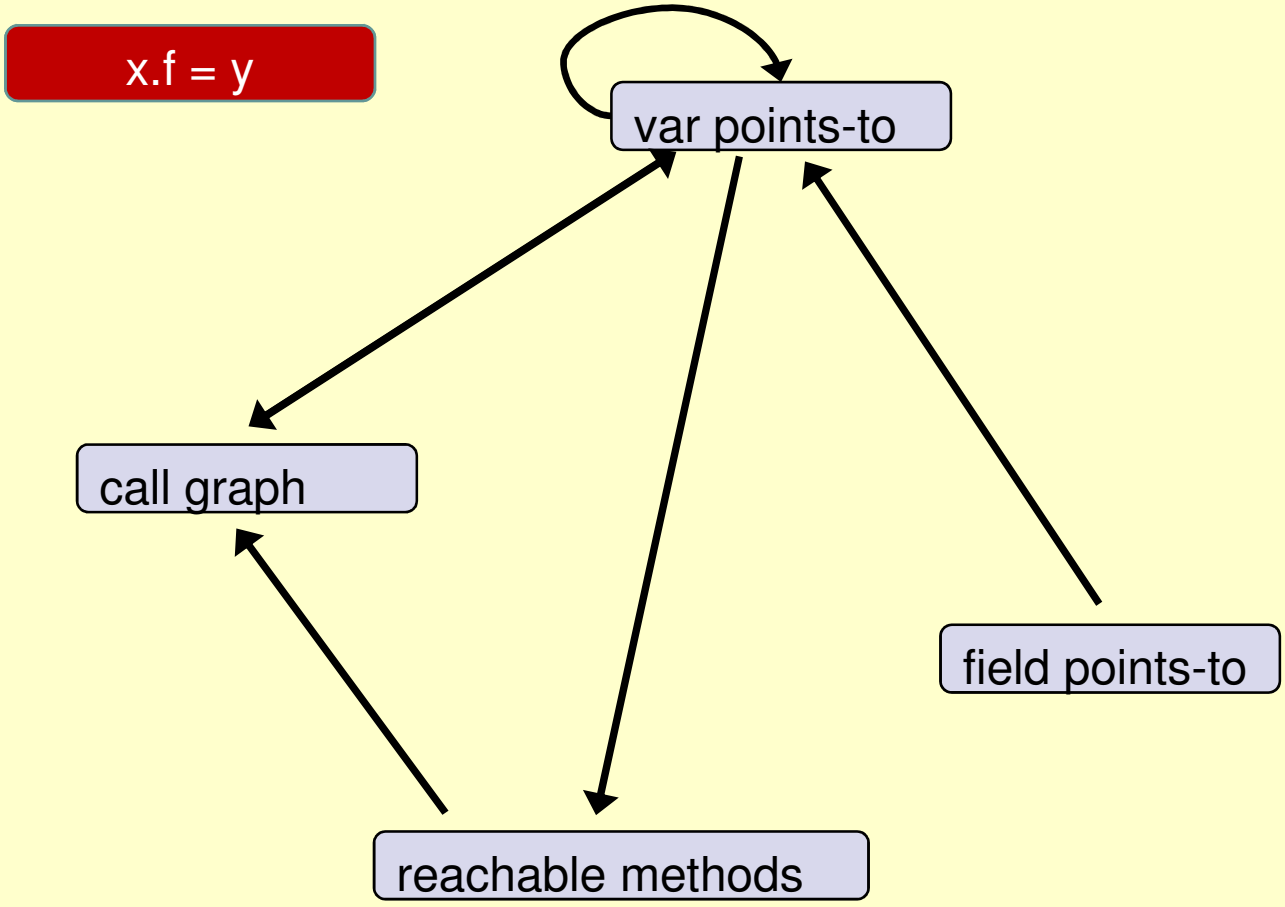


Program Analysis: a Domain of Mutual Recursion

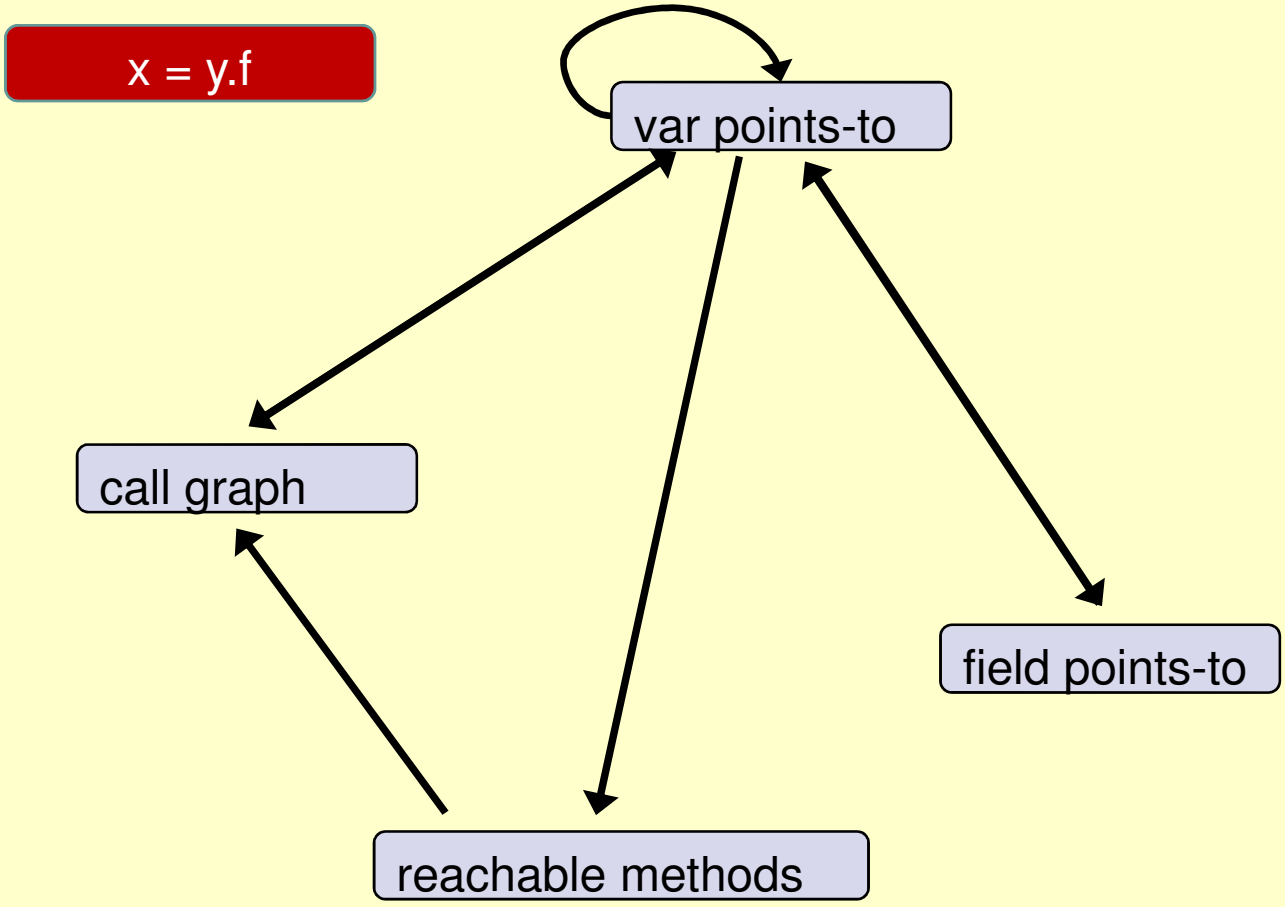




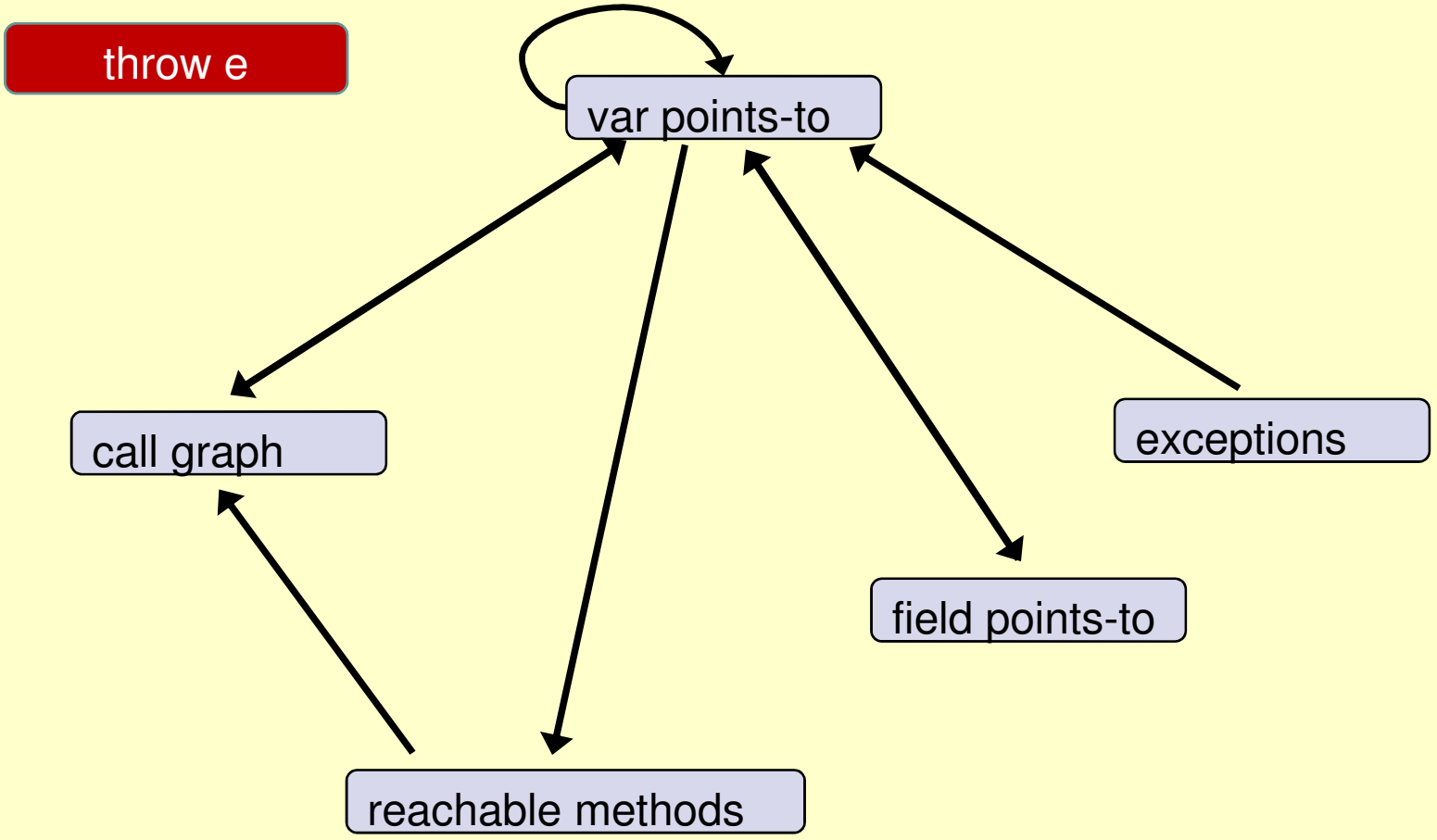
Program Analysis: a Domain of Mutual Recursion



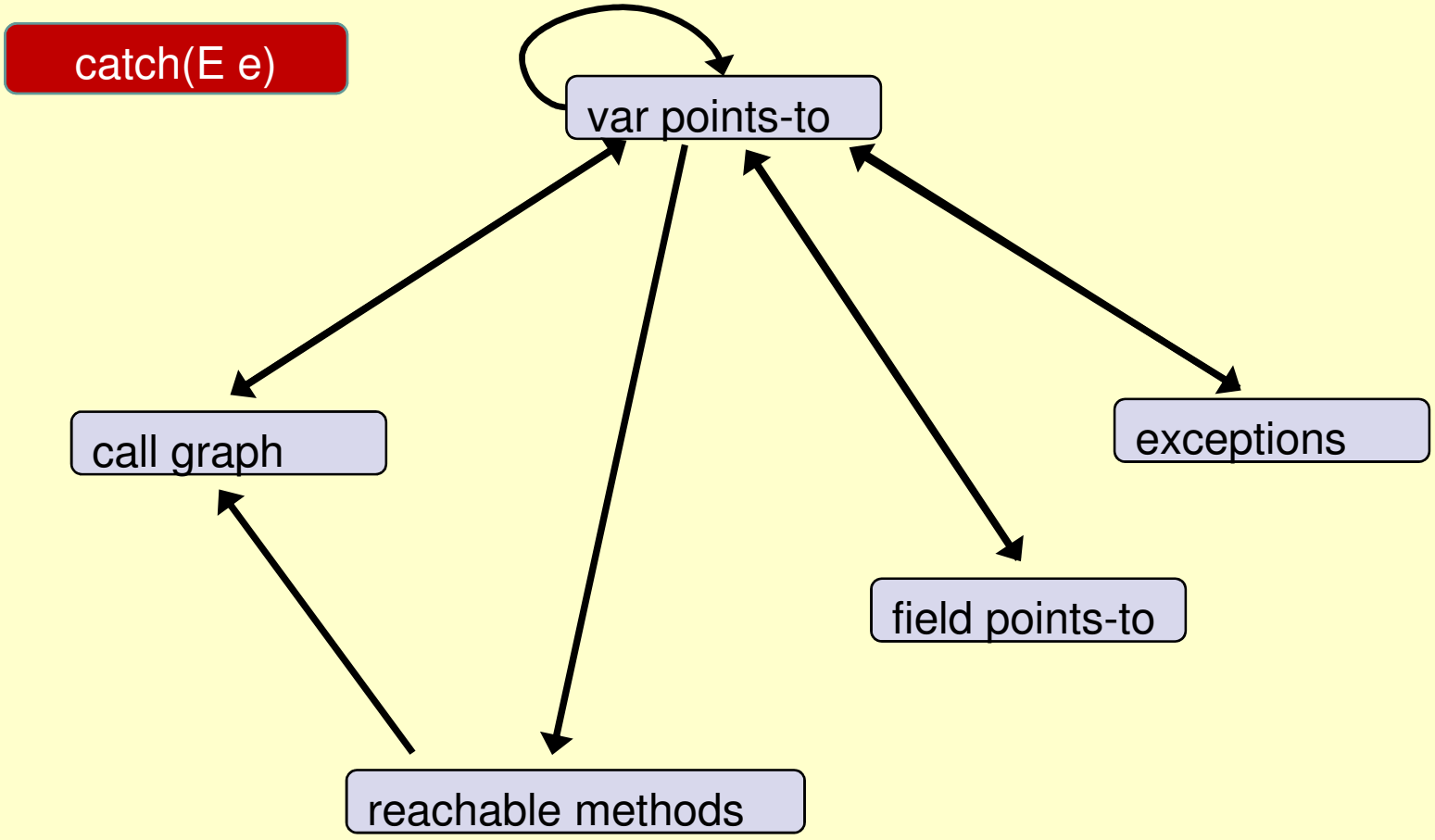
Program Analysis: a Domain of Mutual Recursion



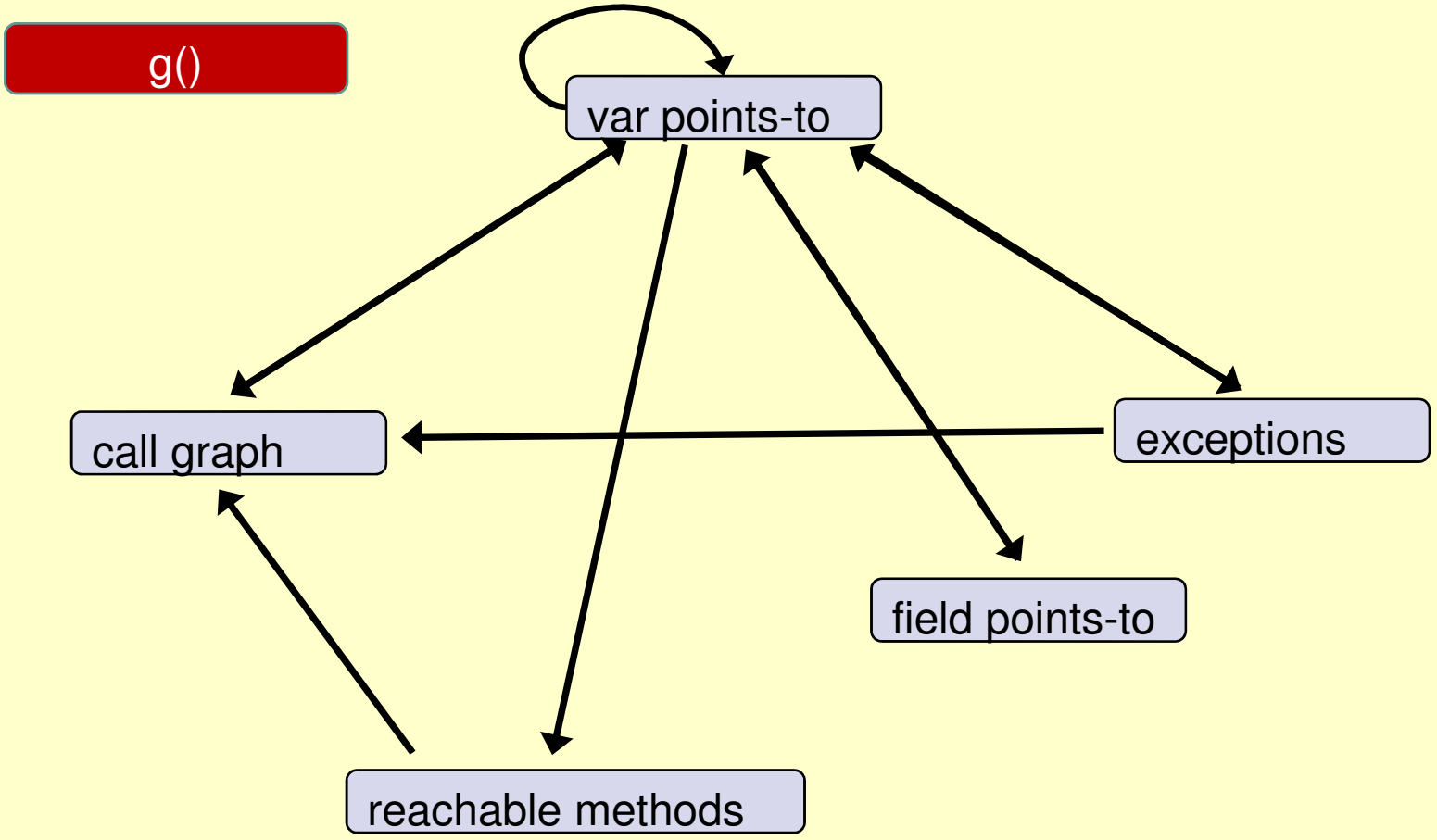
Program Analysis: a Domain of Mutual Recursion



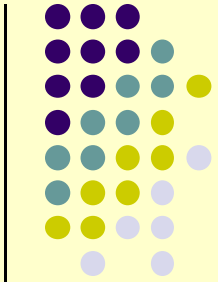
Program Analysis: a Domain of Mutual Recursion



Program Analysis: a Domain of Mutual Recursion

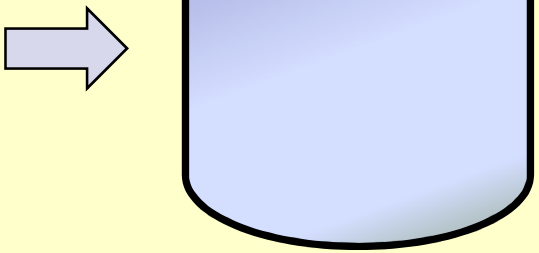


Datalog: Declarative Mutual Recursion

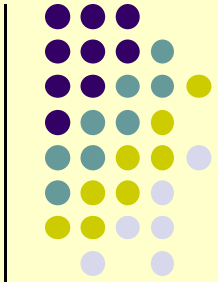


source

```
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()

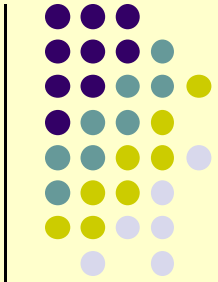
Move

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()

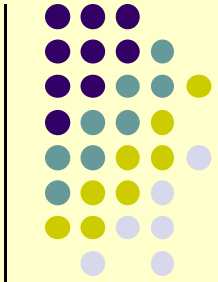
Move

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

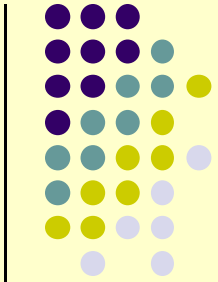
a	b
b	a
c	b

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

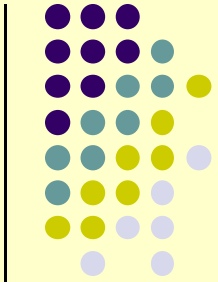
a	b
b	a
c	b

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()

Move

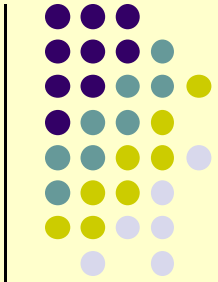
a	b
b	a
c	b

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

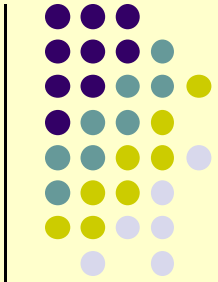
a	b
b	a
c	b

VarPointsTo

join 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

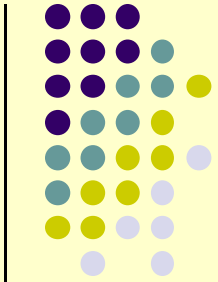
a	b
b	a
c	b

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

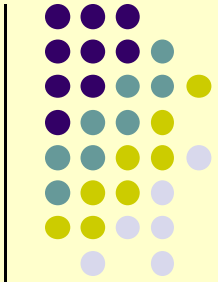
VarPointsTo

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

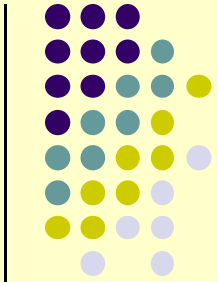
VarPointsTo

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

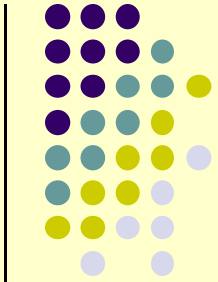
VarPointsTo

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

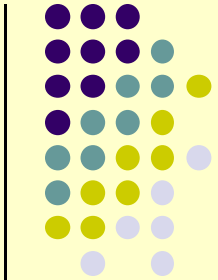
a	b
b	a
c	b

VarPointsTo

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

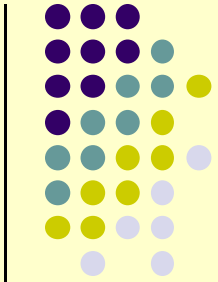
```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

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

Datalog: Properties



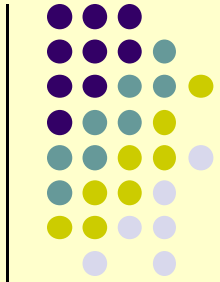
- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - increases algorithm space
    - enables different execution strategies,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

Less programming, more specification



Grand Tour of Interesting Results

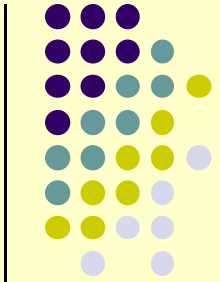
What have we done with this?



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

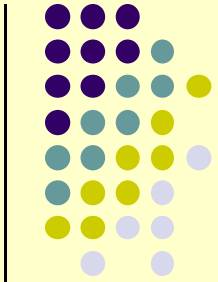
[PLDI'10, POPL'11, CC'13, PLDI'13, PLDI'14]

Expressiveness and Insights



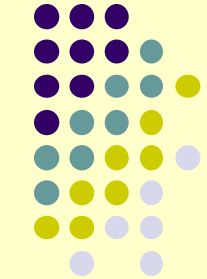
- Greatest benefit of the declarative approach: better algorithms
  - the same algorithms can be described non-declaratively
    - the algorithms are interesting regardless of *how* they are implemented
  - but the declarative formulation was helpful in finding them
    - and in conjecturing that they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

A General Formulation of Context-Sensitive Analyses



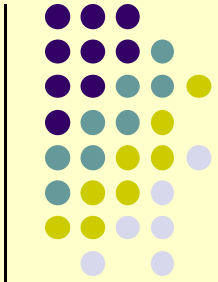
- *Every context-sensitive flow-insensitive analysis there is* (ECSFIATI)
  - ok, almost every
    - most not handled are strictly less sophisticated
  - and also many more than people ever thought
- Also with on-the-fly call-graph construction
- In 9 easy rules!

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.)

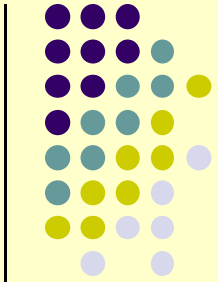
Rule 1: Allocating Objects (Alloc)



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

obj: var = new Something();

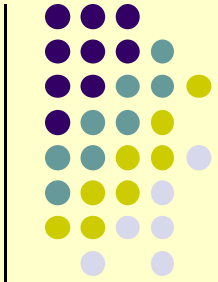
Rule 2: Variable Assignment (Move)



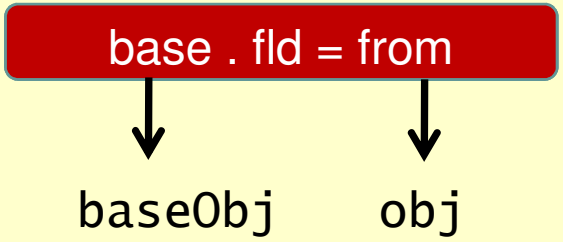
```
VarPointsTo(to, ctx, obj, hctx)
<-
  Move(to, from),
  VarPointsTo(from, ctx, obj, hctx).
```

to = from

Rule 3: Object Field Write (Store)

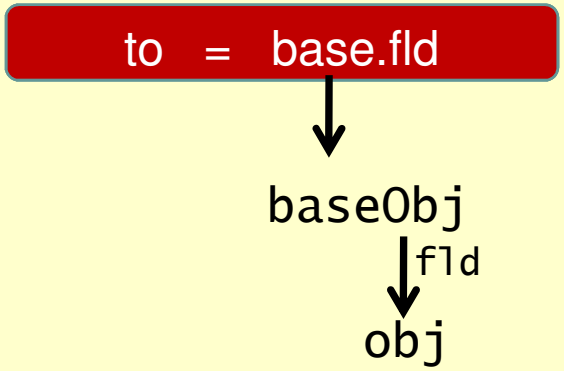


```
FldPointsTo(baseObj, baseHCtx, fld, obj, hctx)
<-
  Store(base, fld, from),
  VarPointsTo(from, ctx, obj, hctx),
  VarPointsTo(base, ctx, baseObj, baseHCtx).
```

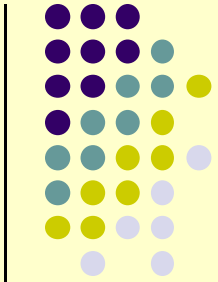


Rule 4: Object Field Read (Load)

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



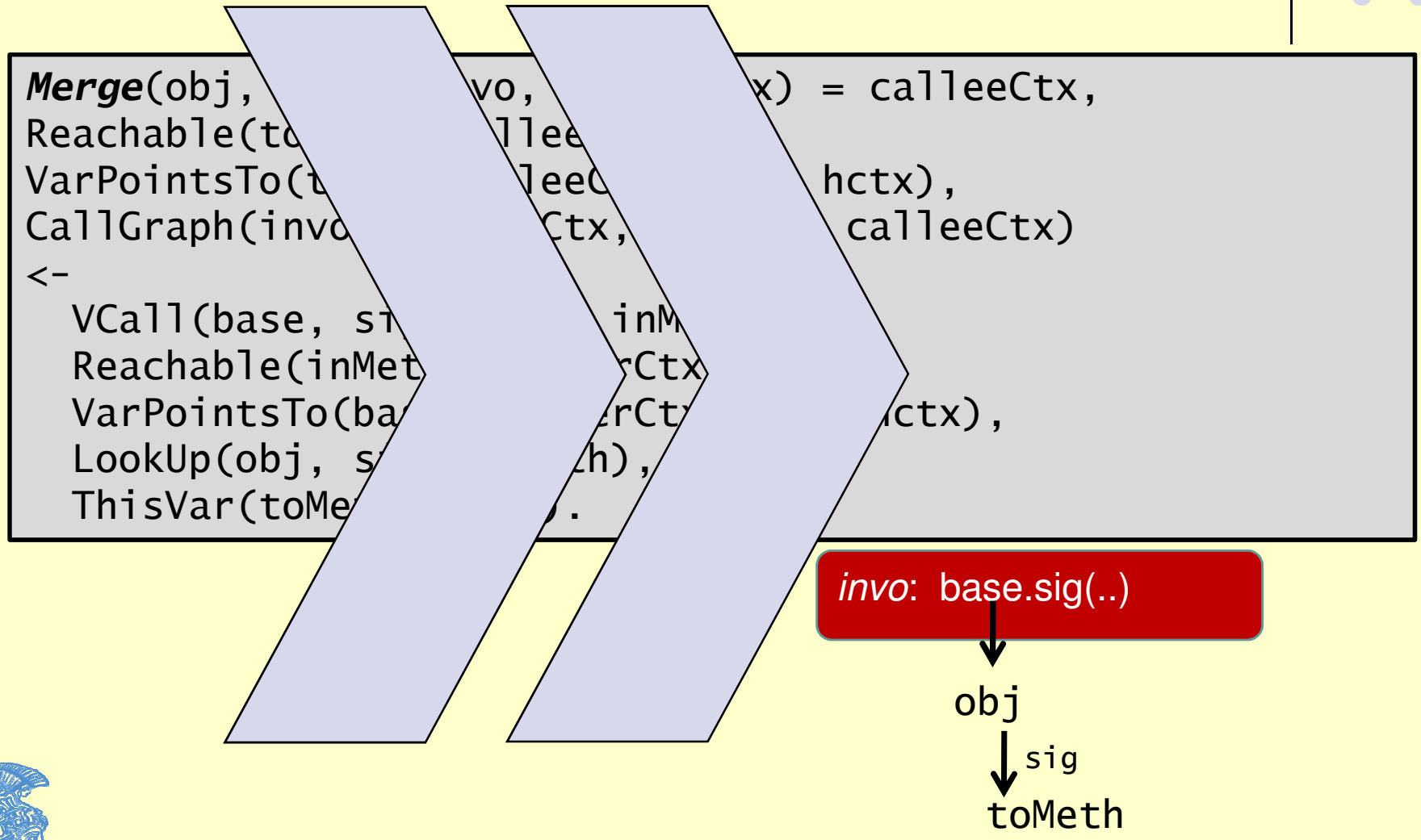
Rule 5: Static Method Calls (SCall)



```
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)



Rule 7: Parameter Passing

```
InterProcAs          ca          from, callerCtx)
<-
  CallGraph(i          lerC          , calleeCtx),
  ActualArg(in          rom)
  FormalArg(met
```

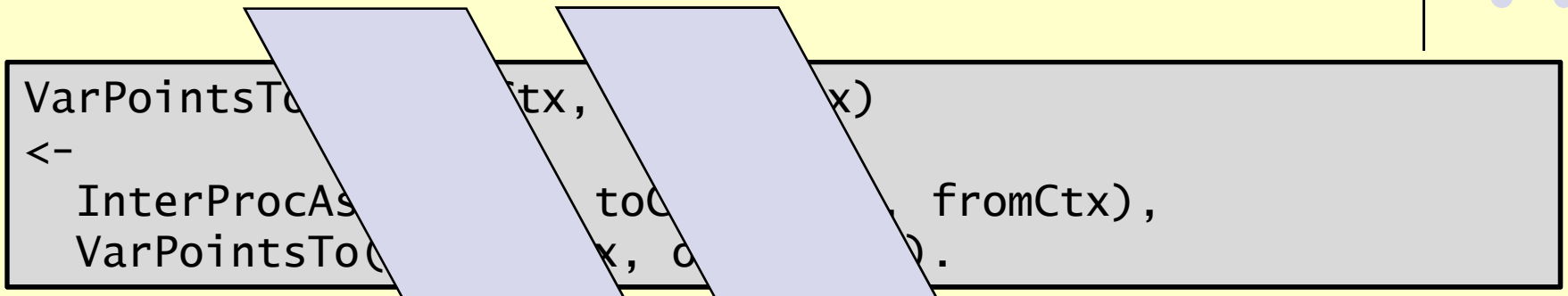
o: n, ..) → meth(.., to, ..)

Rule 8: Return Value Passing

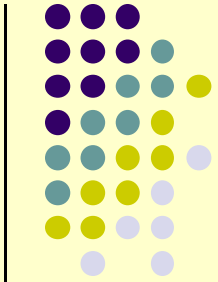
```
InterProcAs          ca          from, calleeCtx)
<-
  CallGraph(t          lerC          , calleeCtx),
  ActualReturn          o),
  FormalReturn(          om).
```

to = meth(..) { .. return from; }

Rule 9: Parameter/Result Passing as Assignment

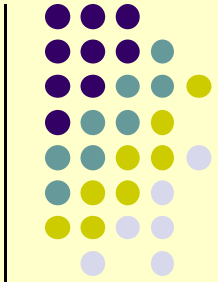


Can Now Express Past Analyses Nicely



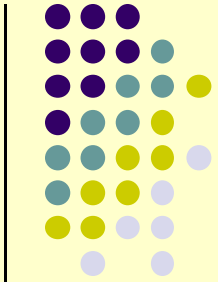
- 1-call-site-sensitive with context-sensitive heap:
  - $Context = HContext = \text{Instr}$
- Functions:
  - ***Record***`(obj, ctx) = ctx`
  - ***Merge***`(obj, hctx, invo, callerCtx) = invo`
  - ***MergeStatic***`(invo, callerCtx) = invo`

Can Now Express Past Analyses Nicely



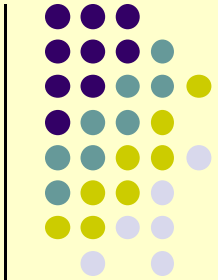
- 1-object-sensitive+heap:
  - $Context = HContext = \text{Instr}$
- Functions:
  - ***Record***(obj, ctx) = ctx
  - ***Merge***(obj, hctx, invo, callerCtx) = obj
  - ***MergeStatic***(invo, callerCtx) = callerCtx

Can Now Express Past Analyses Nicely

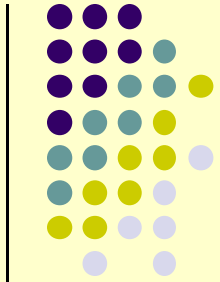


- PADDLE-style 2-object-sensitive+heap:
  - $Context = \text{Instr}^2$, $HContext = \text{Instr}$
- Functions:
  - ***Record***(obj, ctx) = first(ctx)
  - ***Merge***(obj, hctx, invo, callerCtx) =
    pair(obj, first(ctx))
  - ***MergeStatic***(invo, callerCtx) = callerCtx

Lots of Insights and New Algorithms



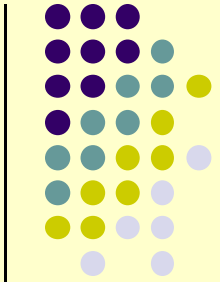
- Discovered that the same name was used for two past algorithms with different behavior
- Proposed a new kind of context (*type-sensitivity*), easily implemented by uniformly tweaking **`Record`**/**`Merge`** functions
- Found connections between analyses in functional/OO languages
- Showed that merging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



Impressive Performance, Implementation Insights

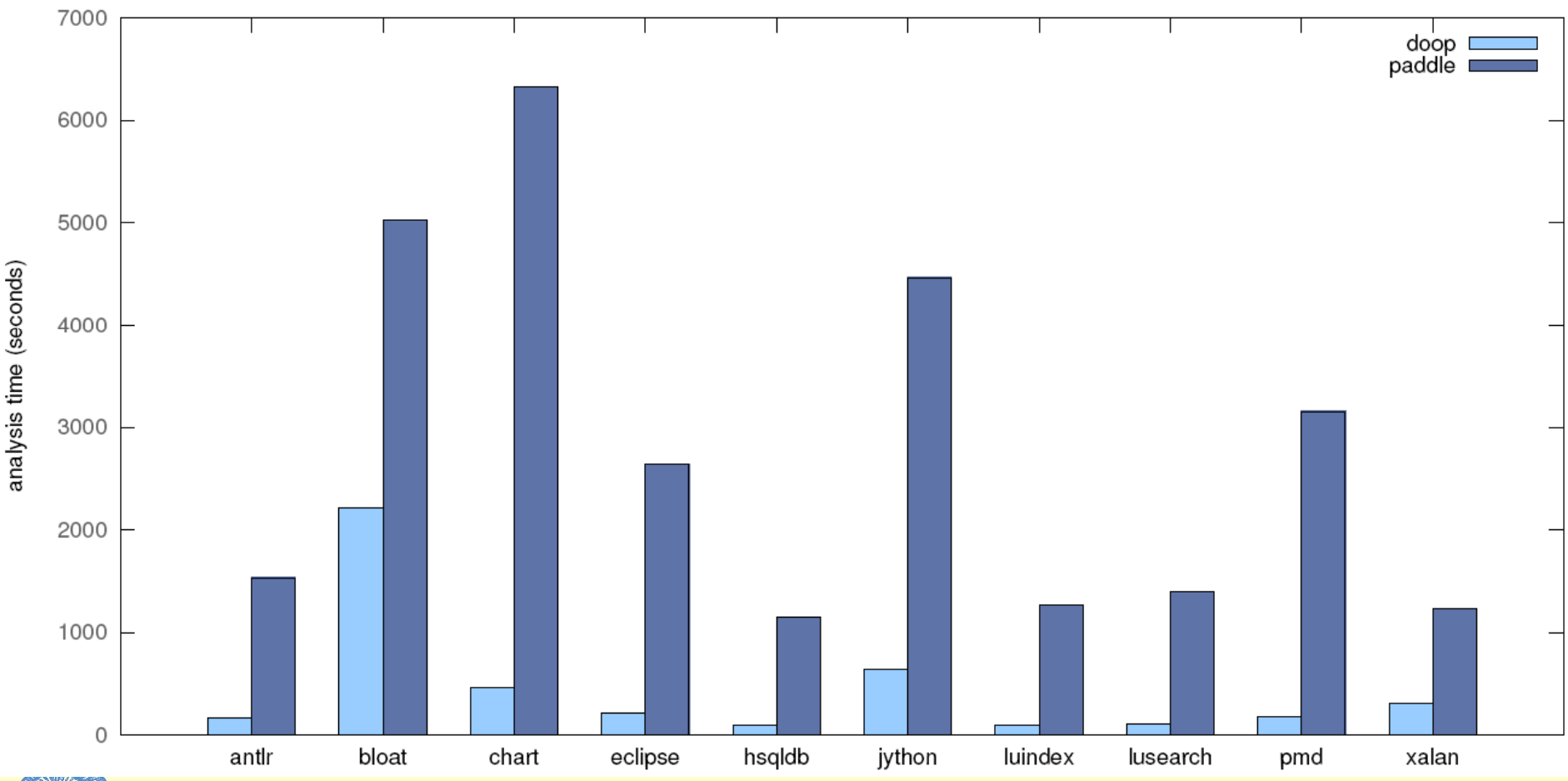
[OOPSLA'09, ISSTA'09]

Impressive Performance

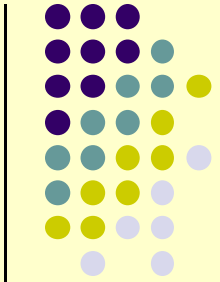


- Compared to Paddle
  - most complete, scalable past framework
    - includes analyses with a context sensitive heap
- Large speedup for fully equivalent results
  - 15.2x faster for 1-obj, 16.3x faster for 1-call, 7.3x faster for 1-call+heap, 6.6x faster for 1-obj+heap
- Large speedup for more precise 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

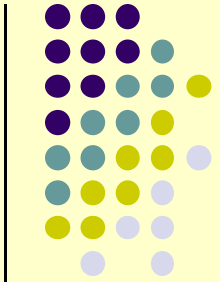


Where Is The Magic?



- 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

Optimization Idea: Optimize Indexing for Semi- Naïve Evaluation



- Datalog rule

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

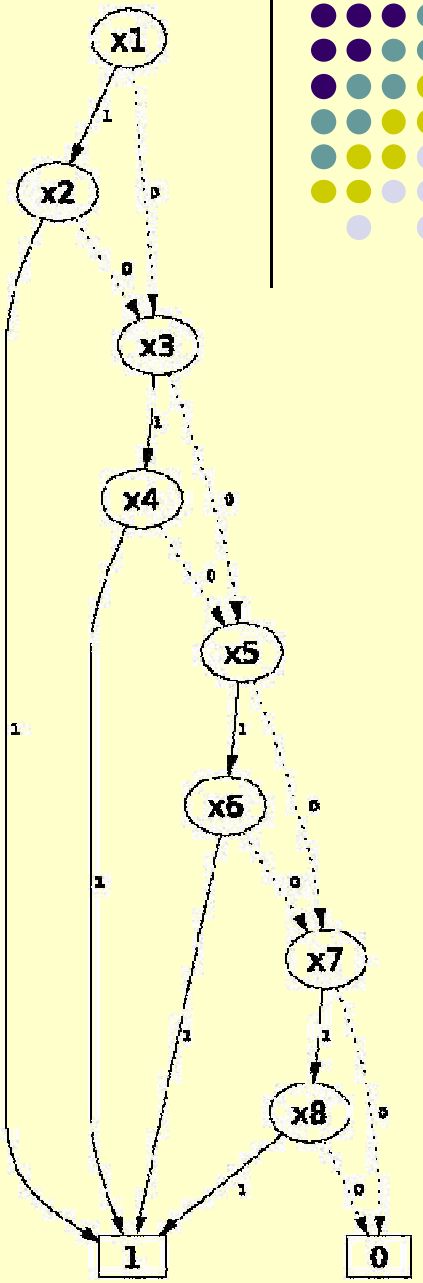
- Semi-Naïve Evaluation

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

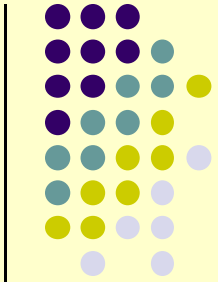
- Ensure the tables are indexed in such way that deltas can bind all index variables
  - `Move` should be indexed from "`from`" to "`to`"
- Harder for multiply recursive rules

No Binary Decision Diagrams (BDDs)

- Scalable precise (context-sensitive) points-to analyses had used BDDs in the past
- We use an explicit representation
- BDDs offer memory efficiency but also overheads
  - traverse 48 links to get a 48-bit tuple
  - cost of normalizing/minimizing

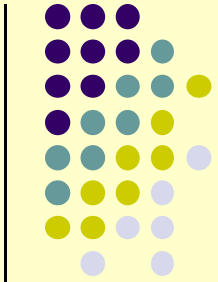


Are BDDs Right For Points-To Analysis?



- We have not found the benefit of BDDs to outweigh the costs
- Relations are reducible, but not clearly extremely regular
  - even though we use BDD variable orderings that have been heavily optimized
    - “impressive results”

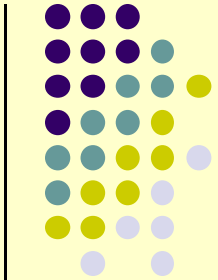
Are BDDs Right For Points-To Analysis?



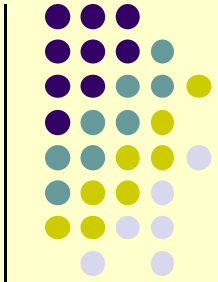
- The Paddle results optimize for speed, size of `VarPointsTo` relation
- But other relations may be large
- For no analysis does the "optimal" BDD ordering simultaneously minimize relations `VarPointsTo`, `FieldsPointsTo`, `CallGraphEdge`
- 30x differences in ratio facts/BDDnodes are common!

BDDs (as currently used in points-to analyses) do not seem to pay off

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!

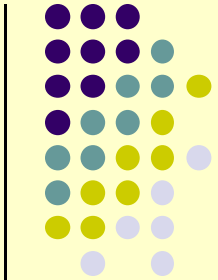


Set-Based Pre-Analysis

a universal optimization technique for flow-insensitive analyses

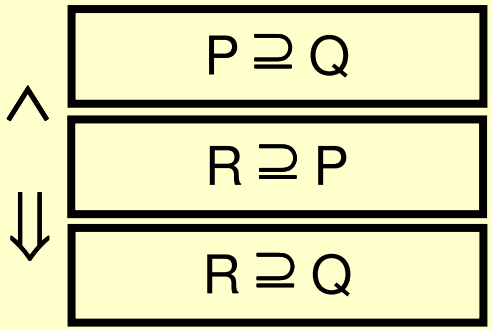
[OOPSLA'13]

Set-Based Pre-Analysis



- Idea: can do much reasoning at the *set level* instead of the *value level*
  - can simplify the program as a result
  - a local transformation
  - think of it as creating a normal form (or IR) for points-to analysis

“hello, world” Example

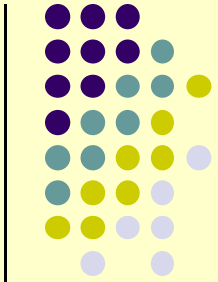



```
p = q;
r = p;
r = q;  // redundant
```

(MOVE elimination)

- Simple subset reasoning
  - statement redundant for *analysis* purposes

“hello, world” Example



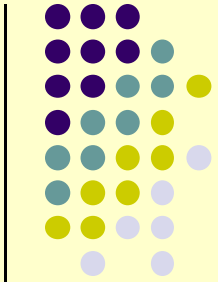
occurring in any order **anywhere** in same method

```
p = q;
r = p;
r = q;  // redundant
```

(MOVE elimination)

- Simple subset reasoning
  - statement redundant for *analysis* purposes
- Rewrite program, eliminate redundant statement
  - an intraprocedural, pattern-based transformation

More Examples



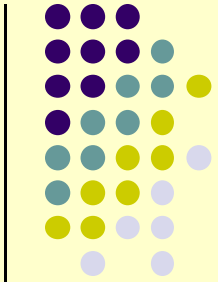
```
r = q;
p.f = r;
p.f = q;  // redundant
```

(STORE elimination)

```
p.f = q;
r = p.f
r = q;    // redundant
```

(another MOVE elimination)

Even More Examples



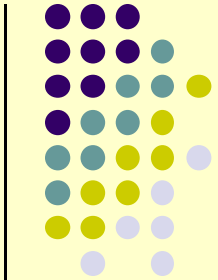
```
r = q;
q = p.f;
r = p.f;    // redundant
```

(LOAD elimination)

```
r = q;
q = p.m();
r = p.m(); // redundant
```

(CALL elimination!!!)

Not Even Close To Done



```
r = q;
q = arr[*];
r = arr[*];    // redundant
```

(ARRAYLOAD elimination—
for array insensitive analyses)

- can apply all previous patterns in combination with array ops, or with static loads, calls, stores, etc.
- transforms apply to fixpoint (one enables others)

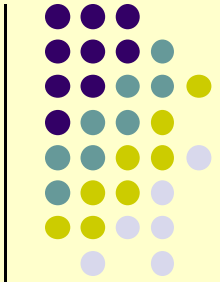
And Also…

- Duplicate variable elimination
  - same as past work using the constraint graph to merge points-to sets
- E.g.,
  - merge vars in same strongly connected component of constraint graph [Faehndrich et al.]
  - merge vars with identical in-flows [Rountev and Chandra, Hardekopf and Lin]
  - merge vars with same dominator [Nasre]

Constraint Graph:

∀
p = q; Q → P

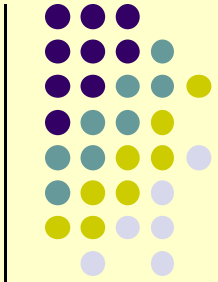
Transform Effect, Pictorially



```
private void rotateRight(java.util.TreeMap$Entry)
          java.util.TreeMap r0;
          java.util.TreeMap$Entry r1, r2, $r3, $r4, $r5, $r6, $r7, $r8, $r9, $r10, $r11;

          r0 := @this: java.util.TreeMap;
          r1 := @param0: java.util.TreeMap$Entry;
          if r1 == null goto label4;

          r2 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left>;
          $r3 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
          r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = $r3;
          $r4 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
          if $r4 == null goto label0;

          $r5 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
          $r5.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = r1;
   label0: $r6 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
          r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r6;
          $r7 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
          if $r7 != null goto label1;

          r0.<java.util.TreeMap: java.util.TreeMap$Entry root> = r2;
          goto label3;
   label1: $r8 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
          $r9 = $r8.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
          if $r9 != r1 goto label2;

          $r10 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
          $r10.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = r2;
             goto label3;
   label2: $r11 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
          $r11.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = r2;
   label3: r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = r1;
          r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = r2;
   label4: return;
```

```
private void rotateRight(java.util.TreeMap$Entry)
          java.util.TreeMap$Entry r2, $r3, $r6, $r9;

          if @param0 == null goto label4;

          r2 = @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left>;
          $r3 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
          @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = $r3;
          if $r3 == null goto label0;

          $r3.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = @param0;
   label0: $r6 = @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
          r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r6;
          if $r6 != null goto label1;

          @this.<java.util.TreeMap: java.util.TreeMap$Entry root> = r2;
          goto label3;
   label1: $r9 = $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
          if $r9 != @param0 goto label2;

          $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = r2;
          goto label3;
   label2: $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = r2;
   label3: r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = @param0;
          @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = r2;
   label4: return;
```

Observations

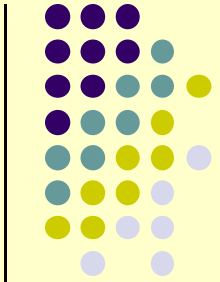


- The reduced program is NOT valid for execution
  - only for flow-insensitive points-to analysis
- Set-based reasoning makes sense since points-to analyses are expressible via subset constraints
  - MOVE elimination follows from MOVE rule in analysis

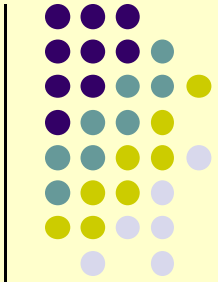
```
p = q;
r = p;
r = q;  // redundant
```

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

So, How Well Does This Work?

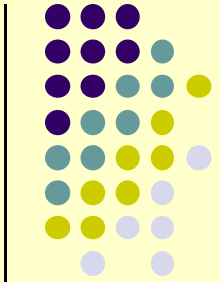


- ***Over many analyses***, DaCapo benchmarks
  - (ctx-insens, 1call, 1call+H, 1obj, 1obj+H, 2obj+H, 2type+H)
- 20% average speedup
  - (median: 20%, max: 110%)
- Eliminates ~30% of local vars
- Decimates (97% elimination!) MOVE instructions
- Eliminates more than 30% of context-sensitive points-to facts



Conclusions, Future Work

Declarative Program Analysis



- Doop has had impact on points-to analysis
  - order-of-magnitude performance improvement
- Several lessons learned
  - new combinations, directions, algorithms
  - algorithmic enhancements, no BDDs
- A lot more analyses are being built on top
  - flow-sensitive, different languages, different contexts, client analyses (escape, may-happen-in-parallel, etc.)