Screaming Fast Declarative Pointer Analysis

http://doop.program-analysis.org

Martin Bravenboer and Yannis Smaragdakis

University of Massachusetts Amherst
University of Oregon

NEPLS
March 5, 2008

what do we do?

what do we do?

- fully declarative pointer analysis
- fast, really fast

overview

what do we do?

- fully declarative pointer analysis
- fast, really fast

how do we do it?

- novel, aggressive optimization
- exposition of indexes

what do we do?

- fully declarative pointer analysis
- fast, really fast

how do we do it?

- novel, aggressive optimization
- exposition of indexes

why do you care?

- fast
- sophisticated, simple
- different

what do we do?

fully declarative pointer analysis
fast, really fast

how do we do it?

novel, aggressive optimization
exposition of indexes

why do you care?

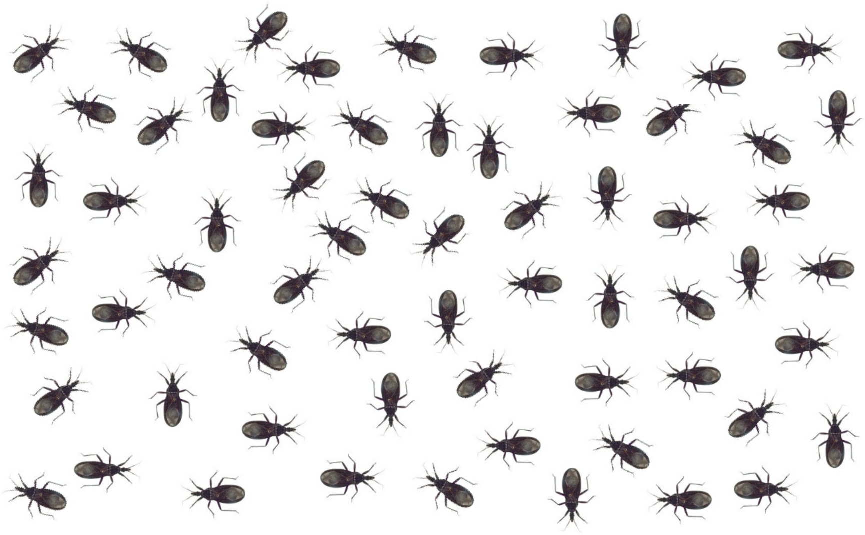
fast
sophisticated, simple
different

why is it relevant?

optimization
understanding, find bugs

program analysis: run faster


NO
UNDER-
STANDING
ANY
TIME



what objects can a variable point to?

program

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

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

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

what objects can a variable point to?

program

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

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

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

points-to

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

what objects can a variable point to?

program

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

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

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

points-to

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

what objects can a variable point to?

program

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

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

A id(A 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()

what objects can a variable point to?

program

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

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

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

points-to

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

context-sensitive points-to

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

pointer analysis: a complex domain

- inclusion-based
- subset-based
- unification-based
- equality-based
- flow-sensitive
- context-sensitive
- k-cfa
- field-based
- field-sensitive
- heap cloning
- context-sensitive heap abstraction

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 in a 10-page pointer analysis paper

```
procedure exhaustive_aliasing(G)
    G: an interprocedural control flow graph (ICFG);
begin
        /* 1. only performed implicitly */
    1.  initialize may_hold with a default value NO;
        create an empty worklist;
    2.  for each node N in G
        2.1  if N is a pointer assignment
                 aliases_intro_by_assignment(N,YES);
        2.2  else if N is a call node
                 aliases_intro_by_call(N,YES);
    3.  while worklist is not empty
        3.1  remove (N, AA, PA) from worklist;
        3.2  if N is a call node
                 alias_at_call_implies(N, AA, PA, YES);
        3.3  else if N is an exit node
                 alias_at_exit_implies(N, AA, PA, YES);
        3.4  else for each M ∈ successor(N)
                 3.4.1  if M is a pointer assignment
                            alias_implies_thru_assign(M,
                            AA, PA, YES);
                 3.4.2  else
                            make_true(M, AA, PA);
end
```

Figure 1: Exhaustive algorithm for *pointer aliasing*

algorithms in a 10-page pointer analysis paper

procedure *exhaustive_aliasing*(G)
    G: …
begin
    1. …
    2. …
    3. …
        alias_at_call_implies(N, AA, PA, YES);
    3.3 **else if** N is an exit node
        alias_at_exit_implies(N, AA, PA, YES);
    3.4 **else for** each $M \in successor(N)$
        3.4.1 **if** M is a pointer assignment
            alias_implies_thru_assign(M, AA, PA, YES);
        3.4.2 **else**
            make_true(M, AA, PA);
end

Figure 1: Exhaustive algorithm for *pointer aliasing*

procedure *incremental_aliasing*(G,N)
    G: an $ICFG$;
    N: a statement to be changed;
begin
    1. falsify the affected aliases, which are either generated at N, or depend on other affected aliases.
    2. update G to reflect the change to statement N;
    3. *worklist=reintroduce_aliases*(G);
    4. *reiterate_worklist*(*worklist*,YES);
end

Figure 2: Incremental aliasing algorithm for handling addition/deletion of a statement

procedure *exhaustive_aliasing*(G)
G:
begin
1.
2.
3.
3.3 **else**
3.4 **else**
end

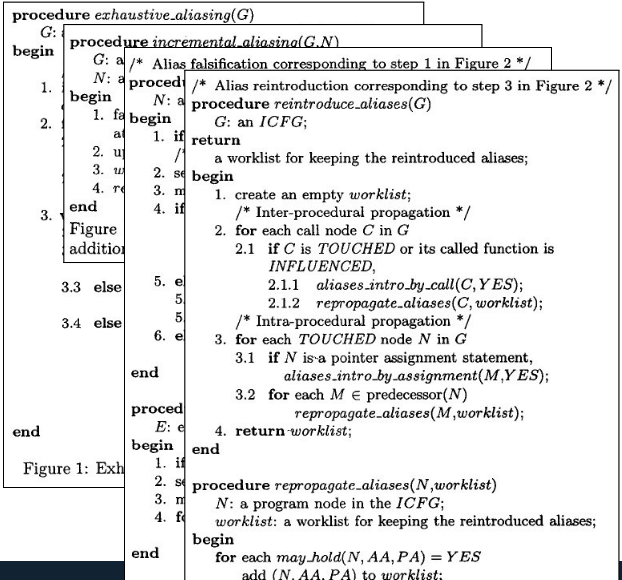
Figure 1: Exh

procedure *incremental_aliasing*(G,N)
G: a
N: a
begin
1. fa
at
2. u
3. u
4. r
end

Figure
additio

/* Alias falsification corresponding to step 1 in Figure 2 */
procedure *naive_falsification*(N)
  N: a statement to be changed;
begin
  1. **if** N is marked *TOUCHED*, **return**;
     /* Falsify aliases at the changed node N */
  2. set all *may_hold*(N, AA, PA) to *NO*;
  3. mark N *TOUCHED*;
  4. **if** N is an exit node
       for each call node C which calls the function containing N;
         naive_falsification(corresponding return of C);
  5. **else if** N is a call node
     5.1 *disable_aliases*(entry of the function called by N);
     5.2 *naive_falsification*(corresponding return of N);
  6. **else for** each $M \in successor(N)$
         naive_falsification(M);
end

procedure *disable_aliases*(E)
  E: entry of the function whose aliases will be disabled;
begin
  1. **if** E is marked *INFLUENCED*, **return**;
  2. set all *may_hold*(E, AA, AA) to *FALSIFIED*;
  3. mark E *INFLUENCED*;
  4. **for** each call node C in function E;
       disable_aliases(entry of the function called by C);
end



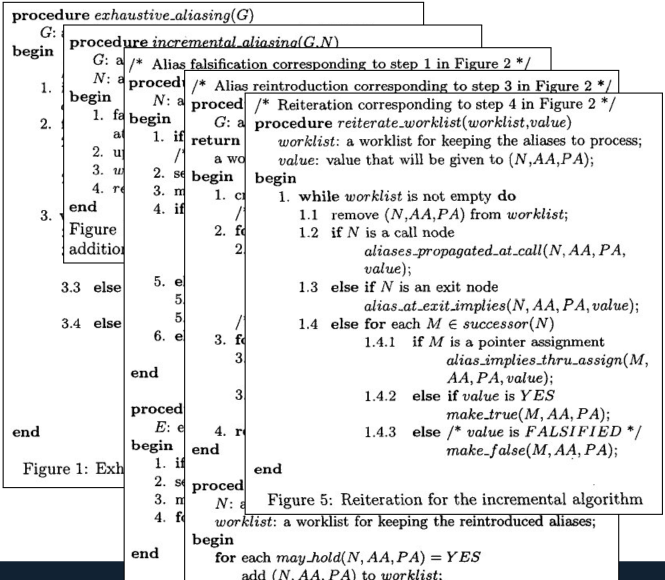
procedure *exhaustive_aliasing*(*G*)

procedure *incremental_aliasing*(*G*,*N*)

/* Alias falsification corresponding to step 1 in Figure 2 */

/* Alias reintroduction corresponding to step 3 in Figure 2 */
procedure *reintroduce_aliases*(*G*)
  G: an *ICFG*;
return
  a worklist for keeping the reintroduced aliases;
begin
  1. create an empty *worklist*;
   /* Inter-procedural propagation */
  2. **for each** call node *C* in *G*
   2.1 **if** *C* is *TOUCHED* or its called function is *INFLUENCED*,
     2.1.1 *aliases_intro_by_call*(*C*, *YES*);
     2.1.2 *repropagate_aliases*(*C*, *worklist*);
   /* Intra-procedural propagation */
  3. **for each** *TOUCHED* node *N* in *G*
   3.1 **if** *N* is a pointer assignment statement,
     aliases_intro_by_assignment(*M*,*YES*);
   3.2 **for each** $M \in$ predecessor(*N*)
     repropagate_aliases(*M*,*worklist*);
  4. **return** *worklist*;
end

procedure *repropagate_aliases*(*N*,*worklist*)
  N: a program node in the *ICFG*;
  worklist: a worklist for keeping the reintroduced aliases;
begin
  for each *may_hold*(*N*, *AA*, *PA*) = *YES*
   add (*N*, *AA*, *PA*) to *worklist*;



procedure *exhaustive_aliasing*(*G*)

procedure *incremental_aliasing*(*G*,*N*)

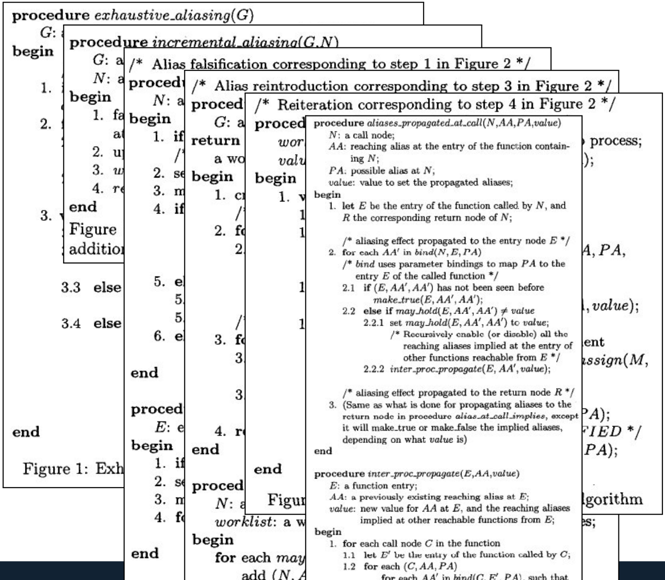
/* Alias falsification corresponding to step 1 in Figure 2 */

/* Alias reintroduction corresponding to step 3 in Figure 2 */

/* Reiteration corresponding to step 4 in Figure 2 */
procedure *reiterate_worklist*(*worklist*,*value*)
  worklist: a worklist for keeping the aliases to process;
  value: value that will be given to (*N*,*AA*,*PA*);
begin
  1. **while** *worklist* is not empty **do**
    1.1 remove (*N*,*AA*,*PA*) from *worklist*;
    1.2 **if** *N* is a call node
        aliases_propagated_at_call(*N*, *AA*, *PA*, *value*);
    1.3 **else if** *N* is an exit node
        alias_at_exit_implies(*N*, *AA*, *PA*, *value*);
    1.4 **else for** each $M \in$ *successor*(*N*)
      1.4.1 **if** *M* is a pointer assignment
          alias_implies_thru_assign(*M*, *AA*, *PA*, *value*);
      1.4.2 **else if** *value* is *YES*
          make_true(*M*, *AA*, *PA*);
      1.4.3 **else** /* *value* is *FALSIFIED* */
          make_false(*M*, *AA*, *PA*);
end

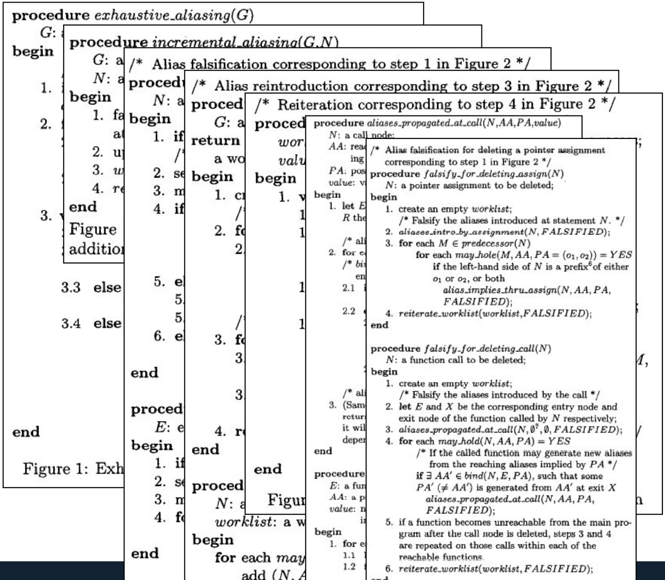
Figure 5: Reiteration for the incremental algorithm

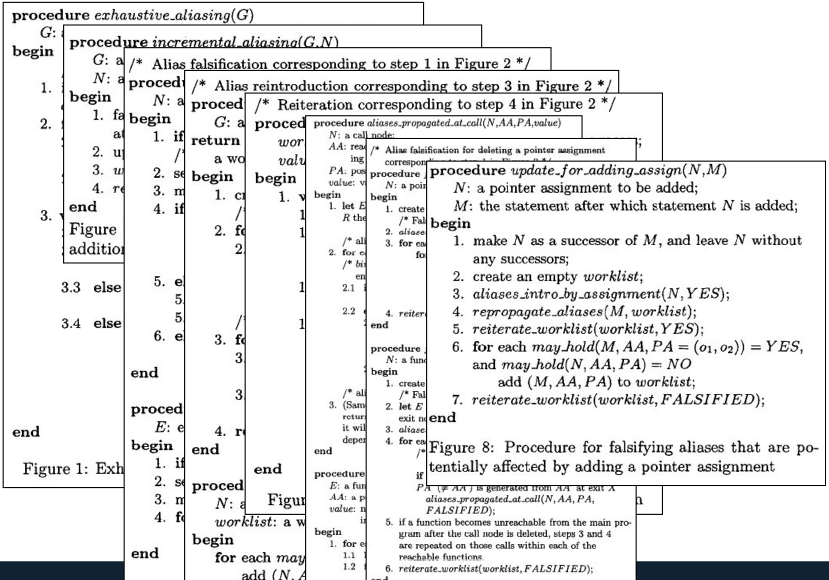
worklist: a worklist for keeping the reintroduced aliases;
begin
  for each *may_hold*(*N*, *AA*, *PA*) = *YES*
    add (*N*, *AA*, *PA*) to *worklist*;



procedure exhaustive_aliasing(G)
procedure incremental_aliasing(G,N)
/* Alias falsification corresponding to step 1 in Figure 2 */
/* Alias reintroduction corresponding to step 3 in Figure 2 */
/* Reiteration corresponding to step 4 in Figure 2 */
procedure aliases_propagated_at_call(N,AA,PA,value)
N: a call node;
AA: reaching alias at the entry of the function containing N;
PA: possible alias at N;
value: value to set the propagated aliases;
begin
1. let E be the entry of the function called by N, and R the corresponding return node of N;
/* aliasing effect propagated to the entry node E */
2. for each AA' in bind(N,E,PA)
/* bind uses parameter bindings to map PA to the entry E of the called function */
2.1 if (E,AA',AA') has not been seen before make_true(E,AA',AA');
2.2 else if may_hold(E,AA',AA') ≠ value
2.2.1 set may_hold(E,AA',AA') to value;
/* Recursively enable (or disable) all the reaching aliases implied at the entry of other functions reachable from E */
2.2.2 inter_proc_propagate(E,AA',value);
/* aliasing effect propagated to the return node R */
3. (Same as what is done for propagating aliases to the return node in procedure alias_at_call_implies, except it will make_true or make_false the implied aliases, depending on what value is)
end
procedure inter_proc_propagate(E,AA,value)
E: a function entry;
AA: a previously existing reaching alias at E;
value: new value for AA at E, and the reaching aliases implied at other reachable functions from E;
begin
1. for each call node C in the function
1.1 let E' be the entry of the function called by C;
1.2 for each (C,AA,PA)
Figure 1: Exh

algorithms in a 10-page pointer analysis paper




procedure exhaustive_aliasing(G)
procedure incremental_aliasing(G,N)
/* Alias falsification corresponding to step 1 in Figure 2 */
/* Alias reintroduction corresponding to step 3 in Figure 2 */
/* Reiteration corresponding to step 4 in Figure 2 */
procedure aliases_propagated_at_call(N,AA,PA,value)
/* Alias falsification for deleting a pointer assignment
procedure update_for_adding_assign(N,M)
N: a pointer assignment to be added;
M: the statement after which statement N is added;
begin
1. make N as a successor of M, and leave N without any successors;
2. create an empty worklist;
3. aliases_intro_by_assignment(N,YES);
4. repropagate_aliases(M,worklist);
5. reiterate_worklist(worklist,YES);
6. for each may_hold(M, AA, PA = (o1, o2)) = YES, and may_hold(N, AA, PA) = NO
add (M, AA, PA) to worklist;
7. reiterate_worklist(worklist, FALSIFIED);
end
Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment
5. if a function becomes unreachable from the main program after the call node is deleted, steps 3 and 4 are repeated on those calls within each of the reachable functions.
6. reiterate_worklist(worklist, FALSIFIED);
Figure 1: Exh

variation points unclear

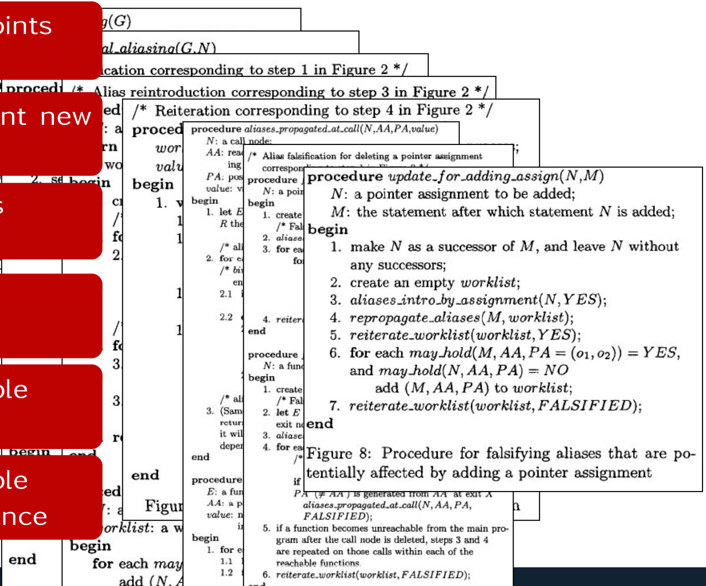
every variant new algorithm

correctness unclear

insights, specifics?

incomparable in precision

incomparable in performance



var points-to

```
x = y
```

var points-to

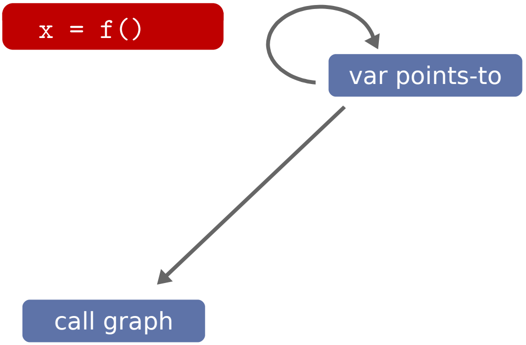
```
x = y
```

program analysis: a domain of mutual recursion

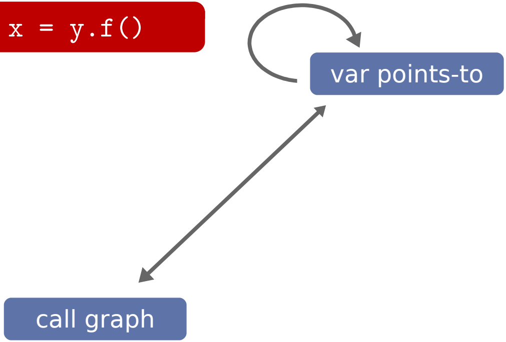
```
x = f()
```

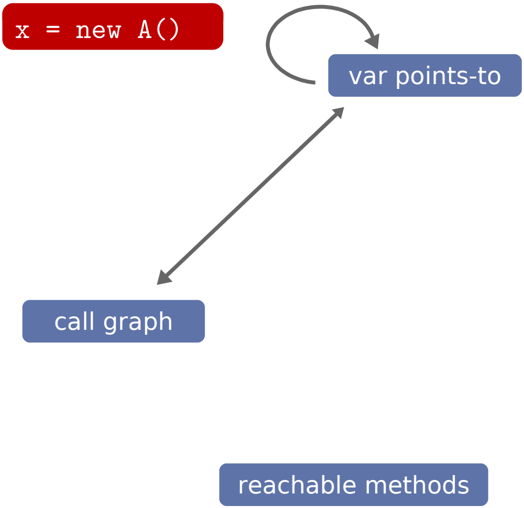
var points-to

call graph

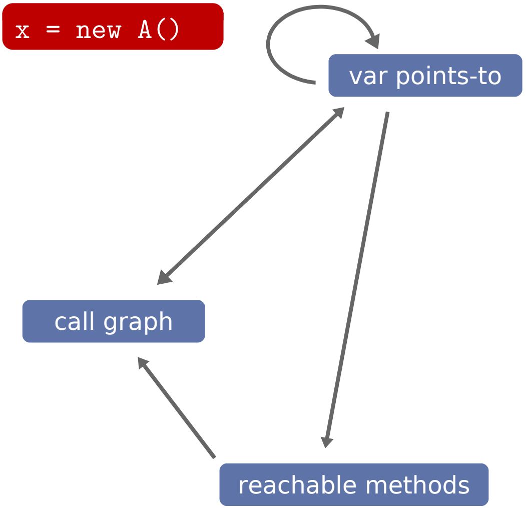

x = f()
var points-to
call graph

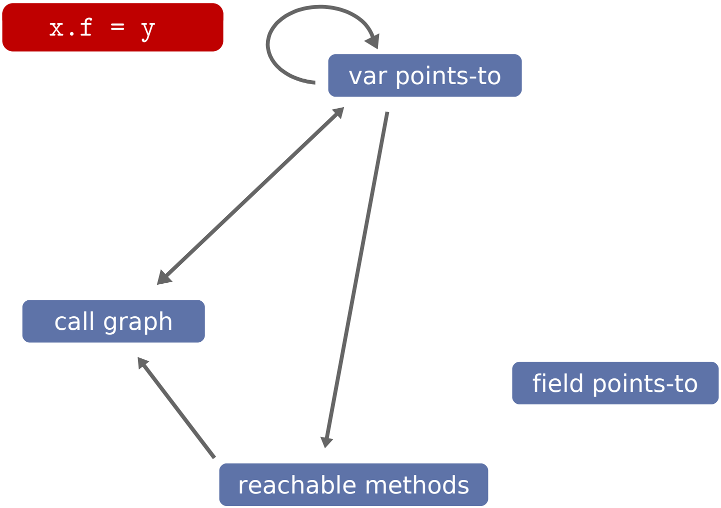
```
x = y.f()
```




x = new A()
var points-to
call graph
reachable methods

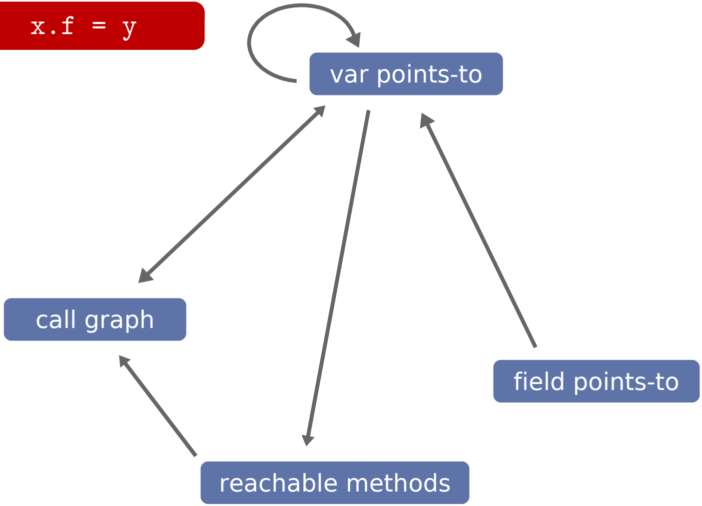
program analysis: a domain of mutual recursion



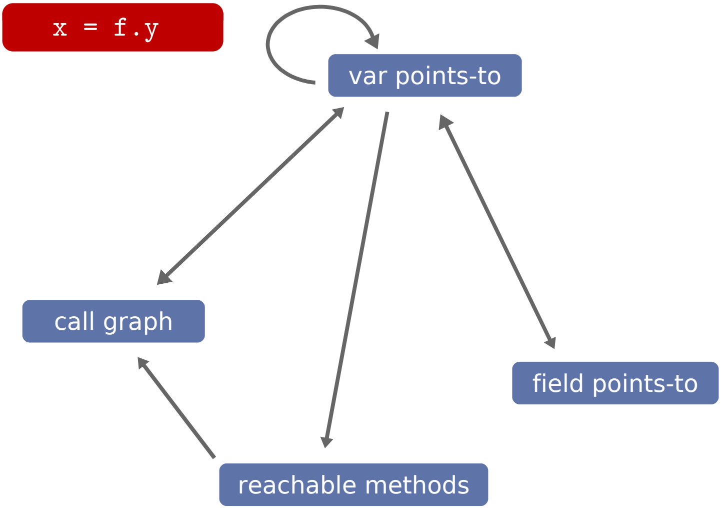

x.f = y
var points-to
call graph
field points-to
reachable methods

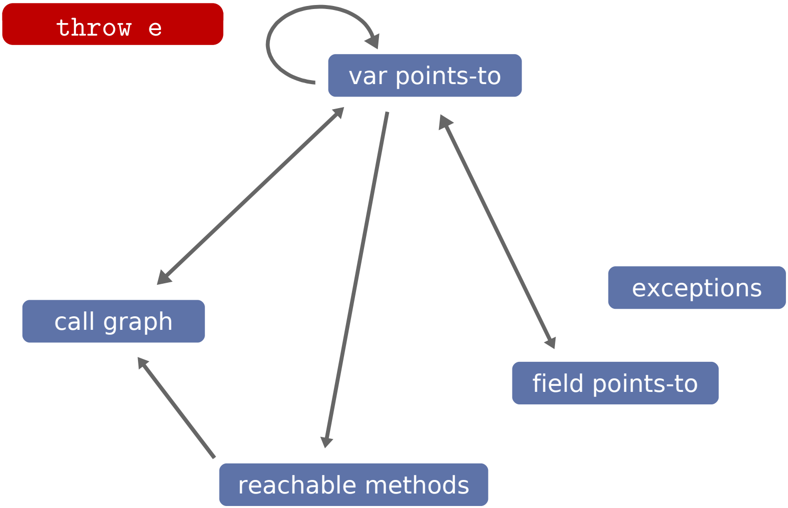
program analysis: a domain of mutual recursion

`x.f = y`

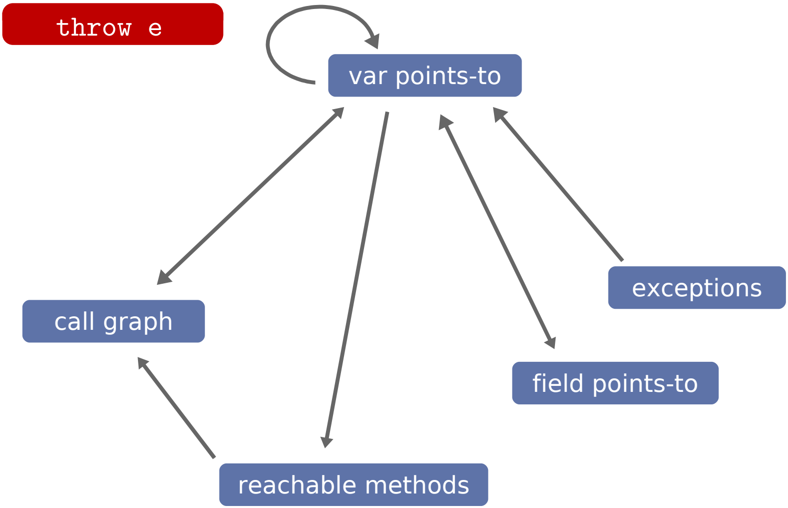


program analysis: a domain of mutual recursion

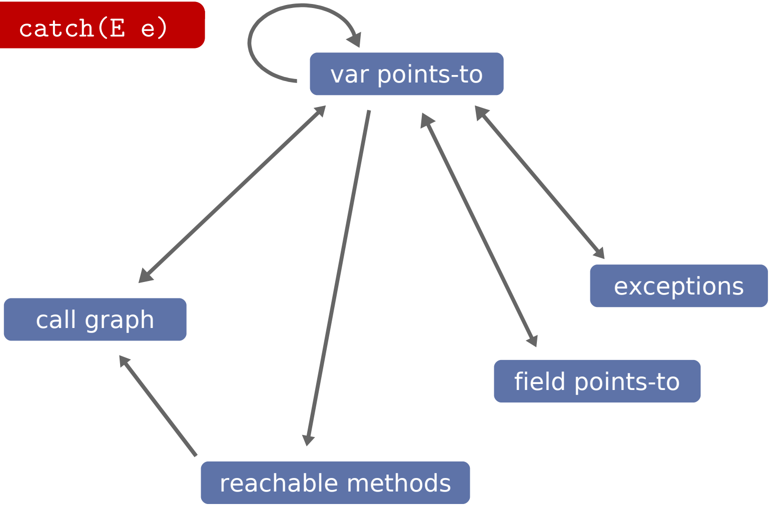


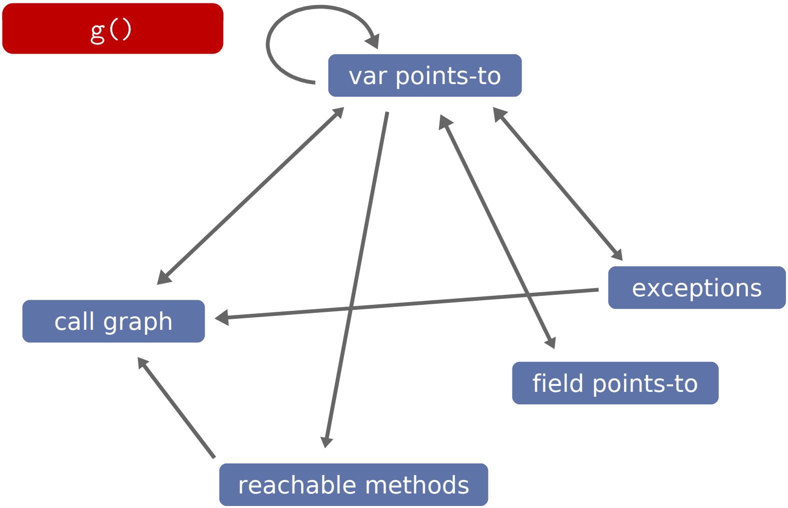

throw e
var points-to
call graph
exceptions
field points-to
reachable methods

program analysis: a domain of mutual recursion



`catch(E e)`


var points-to
call graph
exceptions
field points-to
reachable methods


g()
var points-to
call graph
exceptions
field points-to
reachable methods

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;
```

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

VarPointsTo

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

Assign

b	a
a	b
b	c

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

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

datalog: declarative mutual recursion

source

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

AssignHeapAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

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

limited logic programming

- sql with recursion
  prolog without complex terms (constructors)
- guaranteed termination
  captures PTIME complexity class

purely declarative

- as opposed to prolog
  - conjunction commutative
  - clauses commutative
- enables different execution strategies
- enables more aggressive optimization

writing datalog is less programming, more specification

datalog

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

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

naive evaluation (relational algebra)

VarPointsTo := AssignHeapAllocation

repeat

  tmp := $\pi_{to \rightarrow var, heap}$(VarPointsTo $\bowtie_{from=var}$ Assign)

  VarPointsTo := VarPointsTo $\cup$ tmp

until fixpoint

datalog

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

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

naive evaluation (relational algebra)

VarPointsTo := AssignHeapAllocation

repeat

  tmp := $\pi_{to \rightarrow var, heap}$(VarPointsTo $\bowtie_{from=var}$ Assign)

  VarPointsTo := VarPointsTo $\cup$ tmp

until fixpoint

```
VarPointsTo(var, heap)
Assign(from, to)
```

datalog

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

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

naive evaluation (relational algebra)

VarPointsTo := AssignHeapAllocation

repeat

  tmp := $\pi_{to \to var, heap}$(VarPointsTo $\bowtie_{from=var}$ Assign)

  VarPointsTo := VarPointsTo $\cup$ tmp

until fixpoint

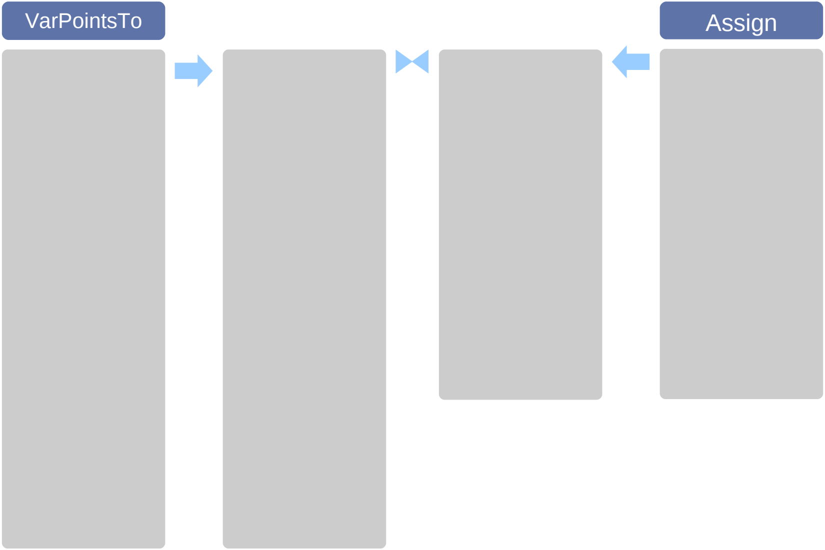
DO loops are so 1950s!

```
VarPointsTo(var, heap)
Assign(from, to)
```

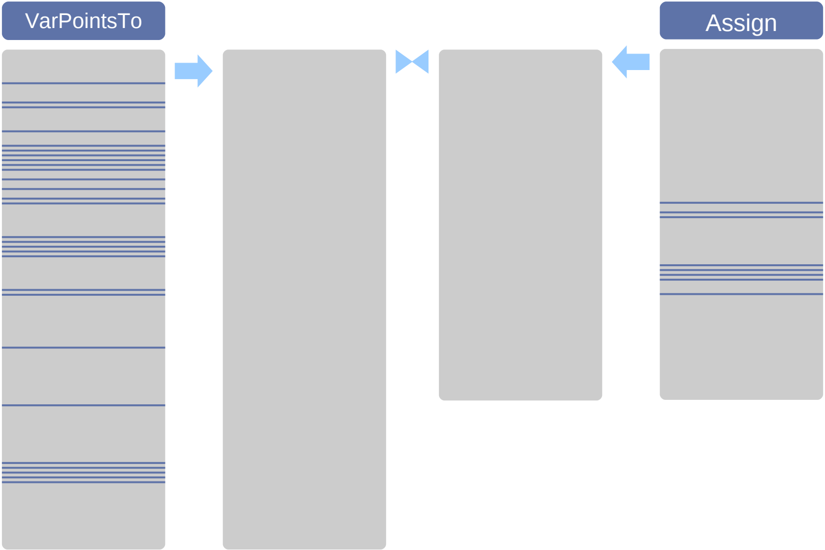
VarPointsTo

Assign

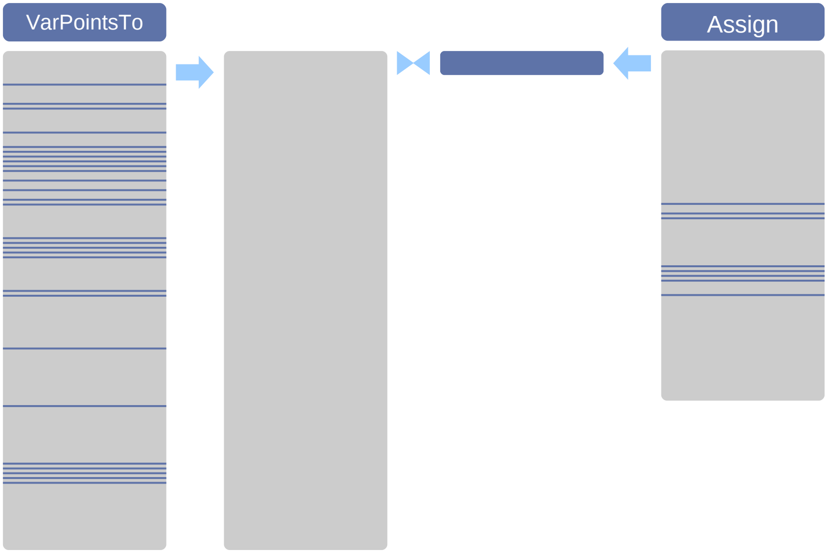
datalog: semi-naive evaluation



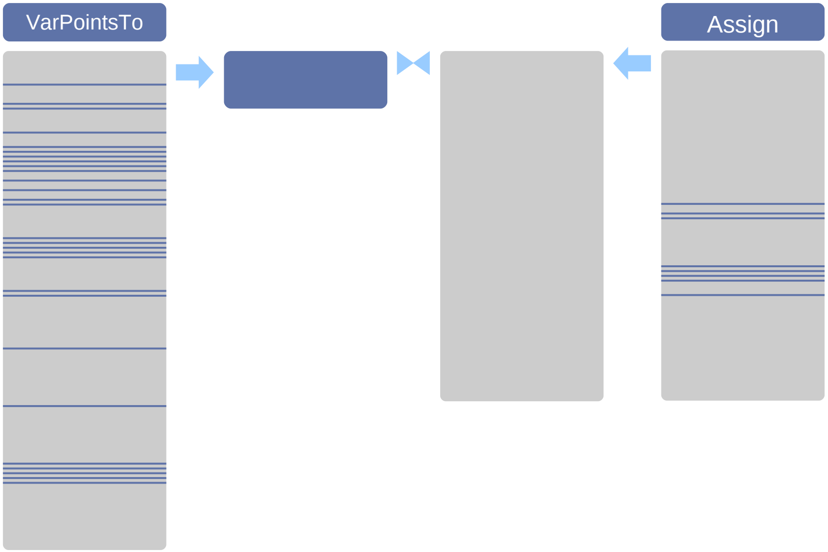
datalog: semi-naive evaluation



datalog: semi-naive evaluation



datalog: semi-naive evaluation



```
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
    StoreInstanceField(?from, ?base, ?signature),
    VarPointsTo(?base, ?baseheap),
    VarPointsTo(?from, ?heap).
```

`StoreInstanceField(?from, ?base, ?signature)`

`?base . ?signature = ?from`

`InstanceFieldPointsTo(?baseheap, ?signature, ?heap)`

field `?signature`
of object `?baseheap`
may point to object `?heap`

```
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
    StoreInstanceField(?from, ?base, ?signature),
    VarPointsTo(?base, ?baseheap),
    VarPointsTo(?from, ?heap).
```

Δ_i`InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-`
    `StoreInstanceField(?from, ?signature, ?base),`
    Δ_{i-1}`VarPointsTo(?base, ?baseheap),`
    `VarPointsTo(?from, ?heap).`

datalog: semi-naive evaluation


```
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
    StoreInstanceField(?from, ?base, ?signature),
    VarPointsTo(?base, ?baseheap),
    VarPointsTo(?from, ?heap).
```

```
Δ_i InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
    StoreInstanceField(?from, ?signature, ?base),
    Δ_{i-1} VarPointsTo(?base, ?baseheap),
    VarPointsTo(?from, ?heap).
```

```
Δ_i InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
    StoreInstanceField(?from, ?signature, ?base),
    VarPointsTo(?base, ?baseheap),
    Δ_{i-1} VarPointsTo(?from, ?heap).
```

pointer analysis: implementation approaches

context-sensitive pointer analysis implementations

paddle

implemented in java + relational algebra + bdd

wala

implemented in java, conventional constraint-based

bddbddb

implemented in datalog + bdd (+ java)

context-sensitive pointer analysis implementations

paddle

implemented in java + relational algebra + bdd

wala

implemented in java, conventional constraint-based

bddbddb

implemented in datalog + bdd (+ java)

not a single
fully declarative
specification

sophisticated pointer analysis fully expressible in datalog

sophisticated pointer analysis fully expressible in datalog

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

unconventional theses

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

Naik: *"All publicly available implementations of k-object sensitive alias analysis ran out of memory on most of our benchmarks for k=1"*

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

Naik: *"All publicly available implementations of k-object sensitive alias analysis ran out of memory on most of our benchmarks for k=1"*

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

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

Naik: *"All publicly available implementations of k-object sensitive alias analysis ran out of memory on most of our benchmarks for k=1"*

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

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

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

Naik: *"All publicly available implementations of k-object sensitive alias analysis ran out of memory on most of our benchmarks for k=1"*

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

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."*

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

semi-naive evaluation: performance $\sim$ size of results

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

semi-naive evaluation: performance $\sim$ size of results

Whaley: *"By using BDDs to represent relations, bddbddb can operate on entire relations at once, instead of iterating over individual tuples."*

sophisticated pointer analysis fully expressible in datalog

explicit representation fits in memory

semi-naive evaluation: performance $\sim$ size of results

sophisticated pointer analysis fully expressible in datalog ✓

explicit representation fits in memory

semi-naive evaluation: performance $\sim$ size of results

- sophisticated pointer analysis fully expressible in datalog ✓
- explicit representation fits in memory ✓
- semi-naive evaluation: performance $\sim$ size of results

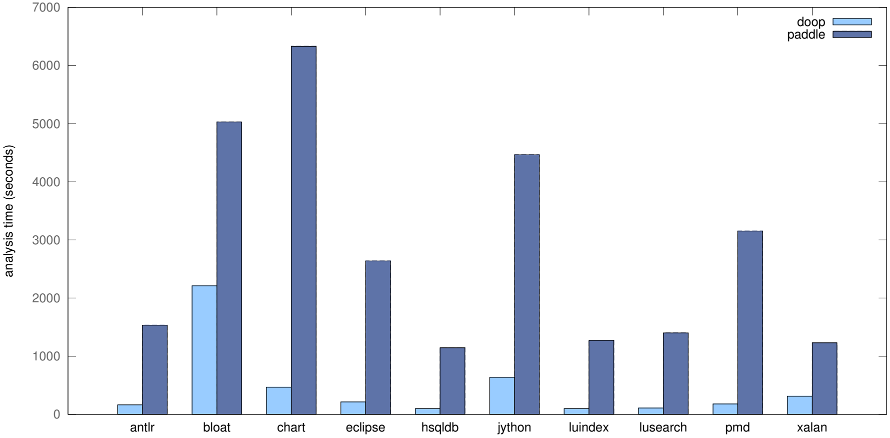
- sophisticated pointer analysis fully expressible in datalog ✓
- explicit representation fits in memory ✓
- semi-naive evaluation: performance $\sim$ size of results ✓

unconventional theses

- sophisticated pointer analysis fully expressible in datalog
- explicit representation fits in memory
- semi-naive evaluation: performance $\sim$ size of results

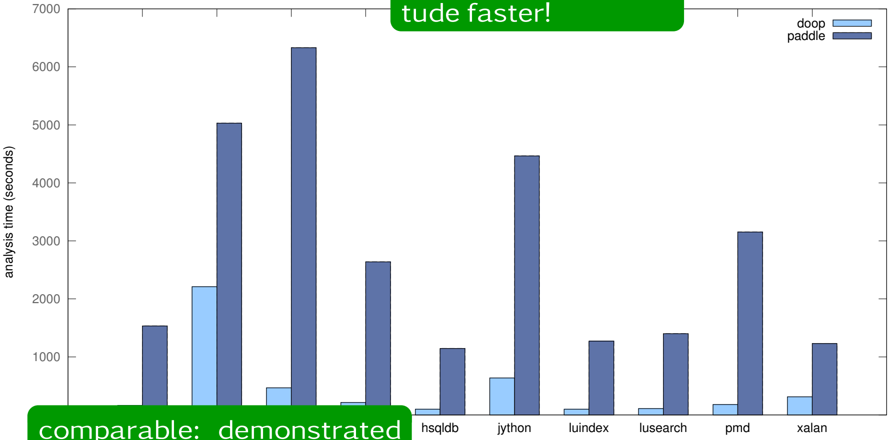
DOOP

benchmark: 1-call-site-sensitive+heap



benchmark: 1-call-site-sensitive+heap

full order of magnitude faster!



comparable: demonstrated equivalence of results!

fully declarative

everything is incremental

efficient architecture for datalog engine

but, not a magic tool!

⇒ novel optimizations targeting recursive logic

yet, relatively easy!

interning

`"<java.lang.Thread:  void <clinit>()>"` → 32-bit int

tuple compression

call-graph edge: invocation × method → 64-bit int

relations are indexes (b-trees)

'index-organized tables'

efficient representation of sparse relations

alternative datastructures for (partially) dense relations

indexes 'exposed' in the language

reverse argument order is index

`VarPointsTo(?var, ?heap)`

LOGICBLOX®

always use index efficiently

- `Assign(?from, ?to), ΔVarPointsTo(?from, ?heap)`
- `Assign(?to, ?from), ΔVarPointsTo(?from, ?heap)`

optimization principles

always use index efficiently

- `Assign(?from, ?to), ΔVarPointsTo(?from, ?heap)`
- `Assign(?to, ?from), ΔVarPointsTo(?from, ?heap)`

never iterate over full views

- `ΔAssign(?to, ?from), VarPointsTo(?from, ?heap)`
- `ΔAssign(?to, ?from), VarPointsTo(?heap, ?from)`

optimization principles

always use index efficiently

- `Assign(?from, ?to), ΔVarPointsTo(?from, ?heap)`
- `Assign(?to, ?from), ΔVarPointsTo(?from, ?heap)`

never iterate over full views

- `ΔAssign(?to, ?from), VarPointsTo(?from, ?heap)`
- `ΔAssign(?to, ?from), VarPointsTo(?heap, ?from)`

never iterate over big input relations

- `StoreInstanceField(?from, ?signature, ?base),`
  `VarPointsTo(?baseheap, ?base),`
  `ΔVarPointsTo(?heap, ?from)`
- we'll get back to that ...

always use index efficiently

- Assign(?from, ?to), ΔVarPointsTo(?from, ?heap)
- Assign(?to, ?from), ΔVarPointsTo(?from, ?heap)

never iterate over full views

- ΔAssign(?to, ?from), VarPointsTo(?from, ?heap)
- ΔAssign(?to, ?from), VarPointsTo(?heap, ?from)

never iterate over big input relations

- StoreInstanceField(?from, ?signature, ?base), VarPointsTo(?baseheap, ?base), ΔVarPointsTo(?heap, ?from)
- we'll get back to that ...

- iterate over delta
- access big relations with index

unoptimized

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

constraints

		Assign	VarPointsTo
1. ΔAssign	2. VarPointsTo	-	?from
1. ΔVarPointsTo	2. Assign	?from	-

optimized

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

optimization: variable ordering

unoptimized

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

constraints

		Assign	VarPointsTo
1. ΔAssign	2. VarPointsTo	-	?from
1. ΔVarPointsTo	2. Assign	?from	-

optimized

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

other join orderings not interesting: pruned immediately

optimization: folding


```
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
  StoreInstanceField(?from, ?base, ?signature),
  VarPointsTo1(?baseheap, ?base),
  VarPointsTo2(?heap, ?from).
```

	Store	$\texttt{VarPointsTo}_1$	$\texttt{VarPointsTo}_2$
1. $\Delta\texttt{VarPointsTo}_1$ 2. StoreInstanceField 3. $\texttt{VarPointsTo}_2$	?base	-	?from
1. $\Delta\texttt{VarPointsTo}_2$ 2. StoreInstanceField 3. $\texttt{VarPointsTo}_1$	?from	?base	-

optimization: folding


```
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
  StoreInstanceField(?from, ?base, ?signature),
  VarPointsTo₁(?baseheap, ?base),
  VarPointsTo₂(?heap, ?from).
```

	Store	$\texttt{VarPointsTo}_1$	$\texttt{VarPointsTo}_2$
1. $\Delta\texttt{VarPointsTo}_1$ 2. StoreInstanceField 3. $\texttt{VarPointsTo}_2$	?base	–	?from
1. $\Delta\texttt{VarPointsTo}_2$ 2. StoreInstanceField 3. $\texttt{VarPointsTo}_1$	?from	?base	–

conflicting constraints! there is no efficient index.

```
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
  StoreInstanceField(?from, ?base, ?signature),
  VarPointsTo₁(?baseheap, ?base),
  VarPointsTo₂(?heap, ?from).
```

fold `StoreInstanceField` and $\texttt{VarPointsTo}_1$ to create an index:

```
InstanceFieldPointsTo(?heap, ?signature, ?baseheap) <-
  StoreHeapInstanceField(?baseheap, ?signature, ?from),
  VarPointsTo(?heap, ?from).

StoreHeapInstanceField(?baseheap, ?signature, ?from) <-
  StoreInstanceField(?from, ?signature, ?base),
  VarPointsTo(?baseheap, ?base).
```

we have just introduced a materialized view!

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

precise exception analysis

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

method invocations: propagated exceptions

```
ThrowPointsTo(?heap, ?callerMethod) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  not exists ExceptionHandler[?heaptype, ?invocation],
  Instruction:Method[?invocation] = ?callerMethod.
```

method invocations: caught exceptions

```
VarPointsTo(?heap, ?param) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?heap, ?tomethod),
  HeapAllocation:Type[?heap] = ?heaptype,
  ExceptionHandler[?heaptype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

declarative program analysis is paying off

declarativeness

precise exception analysis
sophisticated reflection analysis
java references, finalization, threading, etc.

high-level optimization

hand-optimization not difficult
potential automation

automatic incrementalization

naive evaluation would be horrible

automatic memory management

out-of-core analyses

automatic parallelization

very promising for program analysis

Doop

http://doop.program-analysis.org