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
overview

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

why do you care?

fast
sophisticated, simple
different

why is it relevant?

optimization
understanding, find bugs
overview

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
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
program analysis: find bugs
what objects can a variable point to?

program

```java
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?

<table>
<thead>
<tr>
<th>program</th>
<th>points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo() {</td>
<td>foo.a</td>
</tr>
<tr>
<td>a = new A1();</td>
<td>bar.a</td>
</tr>
<tr>
<td>b = id(a);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>void bar() {</td>
<td></td>
</tr>
<tr>
<td>a = new A2();</td>
<td></td>
</tr>
<tr>
<td>b = id(a);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>A id(A a) {</td>
<td></td>
</tr>
<tr>
<td>return a;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
what objects can a variable point to?

**Program**

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

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

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

**Points-to**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.a</td>
<td>new A1()</td>
</tr>
<tr>
<td>bar.a</td>
<td>new A2()</td>
</tr>
<tr>
<td>id.a</td>
<td>new A1(), new A2()</td>
</tr>
</tbody>
</table>
pointer analysis

what objects can a variable point to?

<table>
<thead>
<tr>
<th>program</th>
<th>points-to</th>
</tr>
</thead>
</table>
| void foo() { 
  a = new A1();
  b = id(a);
} |
| foo.a | new A1() |
| bar.a | new A2() |
| id.a | new A1(), new A2() |
| void bar() { 
  a = new A2();
  b = id(a);
} |
| foo.b | new A1(), new A2() |
| bar.b | new A1(), new A2() |
| A id(A a) { 
  return a;
} |
what objects can a variable point to?

**program**

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

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

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

**points-to**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.a</td>
<td>new A1()</td>
</tr>
<tr>
<td>bar.a</td>
<td>new A2()</td>
</tr>
<tr>
<td>id.a</td>
<td>new A1(), new A2()</td>
</tr>
<tr>
<td>foo.b</td>
<td>new A1(), new A2()</td>
</tr>
<tr>
<td>bar.b</td>
<td>new A1(), new A2()</td>
</tr>
</tbody>
</table>

**context-sensitive points-to**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.a</td>
<td>new A1()</td>
</tr>
<tr>
<td>bar.a</td>
<td>new A2()</td>
</tr>
<tr>
<td>id.a (foo)</td>
<td>new A1()</td>
</tr>
<tr>
<td>id.a (bar)</td>
<td>new A2()</td>
</tr>
<tr>
<td>foo.b</td>
<td>new A1()</td>
</tr>
<tr>
<td>bar.b</td>
<td>new A2()</td>
</tr>
</tbody>
</table>
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

1. **Semi-sparse flow-sensitive pointer analysis**
   - 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
   - Full text available: PDF (377.03 KB)
   - Additional Information: full citation, abstract, references, index terms

   **Keywords:** ...
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
Figure 1: Exhaustive algorithm for pointer aliasing

\begin{verbatim}
procedure exhaustive_aliasing(G)
G: procedure incremental_aliasing(G,N)
    G: an ICFG;
    N: a statement to be changed;
begin
1. begin
2. 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

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
\end{verbatim}
procedure exhaustive.aliasing(G)
G: a graph
begin
    procedure incremental.aliasing(G, N)
        G: a graph
        N: a node
        /* 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
                        disable.alises(entry of the function called by N);
                        naive.falsification(corresponding return of N);
                    6. else for each M ∈ successor(N)
                        naive.falsification(M);
            end
        end
    end
end
Figure 1: Exhaustive Alias Falsification

procedure disable.alises(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.alises(entry of the function called by C);
end

Figure 2: Incremental Alias Falsification
procedure exhaustive_aliasing(G)

G: a
begin
procedure incremental_aliasing(G, N)

G: a
N: a
begin
/* Alias falsification corresponding to step 1 in Figure 2 */
procedure reintroduce_aliases(G)

G: an ICFG;
begin
    a worklist for keeping the reintroduced aliases;
    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 rep propagate_aliases(G, 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 ∈ predecessor(N)
          rep propagate_aliases(M, worklist);
    4. return worklist;
end
procedure rep propagate_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;
end

Figure 1: Exhaustive aliasing algorithm.

Figure 2: Incremental aliasing algorithm.
procedure exhaustive_aliasing(G)

begin

G; begin

procedure incremental_aliasing(G,N)

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

G; a

N; a

begin

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

N; a

begin

procedure /* Reiteration corresponding to step 4 in Figure 2 */

G; a

begin

return

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 ∈ 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);

2. for N ∈ worklist do

begin

3. for N ∈ worklist do

begin

4. return

end

end

end

procedure

E: e

end

procedure

N: e

worklist: a worklist for keeping the reintroduced aliases;

begin

begin

for each may_hold(N,AA,PA) = YES

add (N,AA,PA) to worklist;

end

Figure 1: Exhaustive aliasing

Figure 5: Reiteration for the incremental algorithm

end


procedure exhaustive_aliasing(G)

G: a control flow graph
N: a call node

begin
  /* Alias falsification corresponding to step 1 in Figure 2 */
  G: a control flow graph
  N: a call node
  begin
    if false then
      /* Alias reintroduction corresponding to step 3 in Figure 2 */
      G: a control flow graph
      N: a call node
      begin
        if false then
          /* Reiteration corresponding to step 4 in Figure 2 */
          G: a control flow graph
          N: a call node
          begin
            /* Alias propagation at call */
            N: a node at call
            AA: reaching alias at the entry of the function containing
            begin
              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;
                   /* Alias effect propagated to the entry node E */
                   for each AA' in bind(E, E, PA)
                   /* bind uses parameter bindings to map PA to the
                   entry E of the called function */
                   if (E, AA', AA') has not been seen before
                   make_true(E, AA', AA);
                   2.2 else if may_hold(E, AA', AA) # value
                   /* Recursively enable (or disable) all the
                   reaching aliases implied at the entry of
                   other functions reachable from E */
                   inter_proc_propagate(E, AA', value);
                   2.2.2 inter_proc_propagate(E, AA', value);
                   /* Alias 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, except
                   it will make_true or make_false the implied aliases,
                   depending on what value is)
                   3.1 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 may_hold in bind(E, E', PA)
                      add (N, AA, AA') to may_hold;
                      2. for each may_hold in bind(C, C', PA)
                      such that
                      end
                      procedure may_hold(N, AA, AA')
                      N: a node at call
                      AA: a reaching alias at the entry of the function containing
                      AA': a possible alias at N;
                      value: new value for AA at E, and the reaching aliases
                      implied at other reachable functions from E;
                      begin
                      1. let E' be the entry of the function called by C,
                      add (N, AA, AA') to may_hold;
                      end
                    end
                    procedure may_hold(N, AA, AA')
                    N: a node at call
                    AA: a reaching alias at the entry of the function containing
                    AA': a possible alias at N;
                    value: new value for AA at E, and the reaching aliases
                    implied at other reachable functions from E;
                    begin
                    1. let E' be the entry of the function called by C,
                    add (N, AA, AA') to may_hold;
                    end
                  end
                end
              end
            end
          end
        end
      end
    end
  end
end

Figure 1: Exhaustive Aliasing
procedure exhaustive_aliasing(G)
begin
  G; a graph;
  N; an alias node;
begin
  1. if N is a call node
     then return false
     else return true
  2. for each N1 in E
     do if N1 is a call node
         then return false
         else return true
  3. return true
end

procedure incremental_aliasing(G, N)
begin
  G; a graph;
  N; an alias node;
begin
  1. if N is a call node
     then return false
     else return true
  2. for each N1 in E
     do if N1 is a call node
         then return false
         else return true
  3. return true
end

procedure aliases_propagated_at_call(N, AA, PA, value)
begin
  N; a call node;
  AA; a pointer assignment;
  PA; a propagated pointer assignment;
  value; a propagated pointer value;
begin
  1. create an empty worklist;
  2. for each N1 in E
     do if N1 is a call node
         then create a new worklist
         else worklist = N1
  3. repeat
     for each N1 in worklist
     do if N1 is a call node
         then continue
         else add (N1, N, AA, PA, value)
     for each N1 in E
     do if N1 is a call node
         then continue
         else for each N2 in E
               do if N2 is a call node
                   then continue
                   else for each N3 in E
                         do if N3 is a call node
                             then continue
                             else if N3 is a call node
                                 then create a new worklist
                                 else worklist = N3
     until worklist = ∅;
end

procedure false_for_deleting_assign(N)
begin
  N; a pointer assignment to be deleted;
begin
  1. let E be the set of edges
  2. for each N1 in E
     do if N1 is a call node
         then continue
         else start a new worklist
  3. repeat
     for each N1 in worklist
     do if N1 is a call node
         then continue
         else for each N2 in E
               do if N2 is a call node
                   then continue
                   else for each N3 in E
                         do if N3 is a call node
                             then continue
                             else if N3 is a call node
                                 then create a new worklist
                                 else worklist = N3
     until worklist = ∅;
end

procedure false_for_deleting_call(N)
begin
  N; a function call to be deleted;
begin
  1. let E be the set of edges
  2. for each N1 in E
     do if N1 is a call node
         then continue
         else start a new worklist
  3. repeat
     for each N1 in worklist
     do if N1 is a call node
         then continue
         else for each N2 in E
               do if N2 is a call node
                   then continue
                   else for each N3 in E
                         do if N3 is a call node
                             then continue
                             else if N3 is a call node
                                 then create a new worklist
                                 else worklist = N3
     until worklist = ∅;
end

procedure false_for_introduction(N)
begin
  N; a pointer assignment;
begin
  1. let E be the set of edges
  2. for each N1 in E
     do if N1 is a call node
         then continue
         else start a new worklist
  3. repeat
     for each N1 in worklist
     do if N1 is a call node
         then continue
         else for each N2 in E
               do if N2 is a call node
                   then continue
                   else for each N3 in E
                         do if N3 is a call node
                             then continue
                             else if N3 is a call node
                                 then create a new worklist
                                 else worklist = N3
     until worklist = ∅;
end

procedure false_for_reintroduction(N)
begin
  N; a pointer assignment;
begin
  1. let E be the set of edges
  2. for each N1 in E
     do if N1 is a call node
         then continue
         else start a new worklist
  3. repeat
     for each N1 in worklist
     do if N1 is a call node
         then continue
         else for each N2 in E
               do if N2 is a call node
                   then continue
                   else for each N3 in E
                         do if N3 is a call node
                             then continue
                             else if N3 is a call node
                                 then create a new worklist
                                 else worklist = N3
     until worklist = ∅;
end

procedure false_for_reiteration(N)
begin
  N; a pointer assignment;
begin
  1. let E be the set of edges
  2. for each N1 in E
     do if N1 is a call node
         then continue
         else start a new worklist
  3. repeat
     for each N1 in worklist
     do if N1 is a call node
         then continue
         else for each N2 in E
               do if N2 is a call node
                   then continue
                   else for each N3 in E
                         do if N3 is a call node
                             then continue
                             else if N3 is a call node
                                 then create a new worklist
                                 else worklist = N3
     until worklist = ∅;
end

Figure 1: Exhaustive Alias Analysis

Figure 2: Incremental Alias Analysis

procedures in a 10-page pointer analysis paper

Figure 1: Exhaustive aliasing.

```plaintext
procedure exhaustive_aliasing(G)
begin
  procedure incremental_aliasing(G, N)
  begin
    G: a graph
    N: a node
    begin
      /* Alias falsification corresponding to step 1 in Figure 2 */
    procedure aliases_propagated_at_call(N, AA, PA, value)
    begin
      /* Alias reintroduction corresponding to step 3 in Figure 2 */
    procedure update_for_adding_assign(N, M)
    begin
      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, FALSEIFIED)
    end
  end
end
```
variation points
unclear
every variant new
algorithm
correctness
unclear
insights,
specifics?
incomparable
in precision
incomparable
in performance
program analysis: a domain of mutual recursion
program analysis: a domain of mutual recursion

\[ x = y \]

\text{var points-to}
x = y

var points-to
$x = f()$
Program analysis: a domain of mutual recursion

- $x = f()$
- var points-to
- call graph
\( x = y.f() \)
x = new A()

var points-to

call graph

reachable methods
x = new A()
$x.f = y$

- call graph
- var points-to
- reachable methods
- field points-to
program analysis: a domain of mutual recursion

\[ x.f = y \]

- call graph
- var points-to
- field points-to
- reachable methods
\[ x = f.y \]
throw e

var points-to

call graph

reachable methods

exceptions

field points-to
program analysis: a domain of mutual recursion

throw e

var points-to

call graph

exceptions

field points-to

reachable methods
catch(E e)

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

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

<table>
<thead>
<tr>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = new A();</td>
</tr>
<tr>
<td>b = new B();</td>
</tr>
<tr>
<td>c = new C();</td>
</tr>
<tr>
<td>a = b;</td>
</tr>
<tr>
<td>b = a;</td>
</tr>
<tr>
<td>c = b;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AssignHeapAllocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assign</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

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(?var, ?heap) <-
AssignHeapAllocation(?var, ?heap).
VarPointsTo(?to, ?heap) <-
Assign(?from, ?to),
VarPointsTo(?from, ?heap).
datalog: declarative mutual recursion

source

\[
a = \text{new } A(); \\
b = \text{new } B(); \\
c = \text{new } C(); \\
a = b; \\
b = a; \\
c = b;
\]

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

VarPointsTo(?to, ?heap) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?heap).
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;

---

### AssignHeapAllocation

<table>
<thead>
<tr>
<th>var</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>

### Assign

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

### VarPointsTo

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

VarPointsTo(?to, ?heap) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?heap).
### datalog: declarative mutual recursion

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

#### AssignHeapAllocation
```plaintext
<table>
<thead>
<tr>
<th>var</th>
<th>allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>new A()</td>
</tr>
<tr>
<td>b</td>
<td>new B()</td>
</tr>
<tr>
<td>c</td>
<td>new C()</td>
</tr>
</tbody>
</table>
```

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

#### Assign
```plaintext
<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
```

#### VarPointsTo
```plaintext
VarPointsTo(?to, ?heap) <-
    Assign(?from, ?to),
    VarPointsTo(?from, ?heap).
```
source
\[
\begin{align*}
    a &= \text{new } A(); \\
    b &= \text{new } B(); \\
    c &= \text{new } C(); \\
    a &= b; \\
    b &= a; \\
    c &= b;
\end{align*}
\]

AssignHeapAllocation
\[
\begin{array}{|l|}
\hline
    a & \text{new } A() \\
    b & \text{new } B() \\
    c & \text{new } C() \\
\hline
\end{array}
\]

Assign
\[
\begin{array}{|l|}
\hline
    b & a \\
    a & b \\
    b & c \\
\hline
\end{array}
\]

VarPointsTo
\[
\begin{array}{|l|}
\hline
    a & \text{new } A() \\
    b & \text{new } B() \\
    c & \text{new } C() \\
\hline
\end{array}
\]

\[
\text{VarPointsTo}(\text{?var, ?heap}) \leftarrow \text{AssignHeapAllocation}(\text{?var, ?heap}).
\]

\[
\text{VarPointsTo}(\text{?to, ?heap}) \leftarrow \text{Assign}(\text{?from, ?to}),
\]

\[
\text{VarPointsTo}(\text{?from, ?heap}).
\]
source

\[
\begin{align*}
a &= \text{new A()}; \\
b &= \text{new B()}; \\
c &= \text{new C()}; \\
a &= b; \\
b &= a; \\
c &= b;
\end{align*}
\]

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

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

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

\begin{align*}
\text{VarPointsTo}(?\text{var}, ?\text{heap}) & \leftarrow \\
& \text{AssignHeapAllocation}(?\text{heap}, ?\text{var}).
\end{align*}

\begin{align*}
\text{VarPointsTo}(?\text{to}, ?\text{heap}) & \leftarrow \\
& \text{Assign}(?\text{from}, ?\text{to}), \text{VarPointsTo}(?\text{from}, ?\text{heap}).
\end{align*}

naive evaluation (relational algebra)

\begin{align*}
\text{VarPointsTo} & := \text{AssignHeapAllocation} \\
\text{repeat} & \\
& \quad \text{tmp} := \pi_{\text{to} \rightarrow \text{var}, \text{heap}}(\text{VarPointsTo} \bowtie_{\text{from} = \text{var}} \text{Assign}) \\
& \quad \text{VarPointsTo} := \text{VarPointsTo} \cup \text{tmp} \\
\text{until fixpoint}
\end{align*}
**Datalog: Naive Evaluation**

Datalog:

\[
\text{VarPointsTo}(?\text{var}, ?\text{heap}) \leftarrow \\
\quad \text{AssignHeapAllocation}(?\text{heap}, ?\text{var}).
\]

\[
\text{VarPointsTo}(?\text{to}, ?\text{heap}) \leftarrow \\
\quad \text{Assign}(?\text{from}, ?\text{to}), \text{VarPointsTo}(?\text{from}, ?\text{heap}).
\]

Naive Evaluation (relational algebra):

\[
\text{VarPointsTo} := \text{AssignHeapAllocation} \\
\text{repeat} \\
\quad \text{tmp} := \pi_{\text{to} \rightarrow \text{var}, \text{heap}}(\text{VarPointsTo} \bowtie_{\text{from} = \text{var}} \text{Assign}) \\
\quad \text{VarPointsTo} := \text{VarPointsTo} \cup \text{tmp} \\
\text{until fixpoint}
\]

\[
\text{VarPointsTo}(\text{var}, \text{heap}) \\
\text{Assign(}\text{from}, \text{to})
\]
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 \rightarrow var, heap}(\text{VarPointsTo} \Join_{from = var} \text{Assign})
    VarPointsTo := VarPointsTo \cup tmp
until fixpoint
```
datalog: semi-naive evaluation

VarPointsTo

Assign
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).

\[\Delta_i\text{InstanceFieldPointsTo}(\text{?heap}, \text{?signature}, \text{?baseheap}) \leftarrow\]
\[\text{StoreInstanceField}(\text{?from}, \text{?signature}, \text{?base}),\]
\[\Delta_{i-1}\text{VarPointsTo}(\text{?base}, \text{?baseheap}),\]
\[\text{VarPointsTo}(\text{?from}, \text{?heap}).\]
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
  StoreInstanceField(?from, ?base, ?signature),
  VarPointsTo(?base, ?baseheap),
  VarPointsTo(?from, ?heap).

\[\Delta_i\text{InstanceFieldPointsTo}?(\text{?heap}, \text{?signature}, \text{?baseheap}) \leftarrow \]
  StoreInstanceField(?from, ?signature, ?base),
  \[\Delta_{i-1}\text{VarPointsTo}?(\text{?base}, \text{?baseheap}),\]
  VarPointsTo(?from, ?heap).

\[\Delta_i\text{InstanceFieldPointsTo}?(\text{?heap}, \text{?signature}, \text{?baseheap}) \leftarrow \]
  StoreInstanceField(?from, ?signature, ?base),
  VarPointsTo(?base, ?baseheap),
  \[\Delta_{i-1}\text{VarPointsTo}?(\text{?from}, \text{?heap}).\]
context-sensitive pointer analysis implementations

**paddle**

implemented in java + relational algebra + bdd

**wala**

implemented in java, conventional constraint-based

**bddbddd**

implemented in datalog + bdd (+ java)
context-sensitive pointer analysis implementations

paddle
implemented in java + relational algebra + bdd

wala
implemented in java, conventional constraint-based

bdddbdddb
implemented in datalog + bdd (+ java)

not a single fully declarative specification
unconventional theses

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”
unconventional theses

**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, bddbbdddb 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.”
unconventional theses

- sophisticated pointer analysis fully expressible in datalog
- explicit representation fits in memory
- semi-naive evaluation: performance ~ 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

Whaley: “By using BDDs to represent relations, bddbdbdb can operate on entire relations at once, instead of iterating over individual tuples.”
unconventional theses

- sophisticated pointer analysis fully expressible in datalog
- explicit representation fits in memory
- semi-naive evaluation: performance ~ 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
unconventional theses

- 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

Comparative analysis showing full order of magnitude faster!

Comparable: demonstrated equivalence of results!
where is the magic?

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)
always use index efficiently

- Assign(?from, ?to), ∆VarPointsTo(?from, ?heap)
- Assign(?to, ?from), ∆VarPointsTo(?from, ?heap)
always use index efficiently

- \( \text{Assign}(\text{?from}, \text{?to}), \Delta \text{VarPointsTo}(\text{?from}, \text{?heap}) \)
- \( \text{Assign}(\text{?to}, \text{?from}), \Delta \text{VarPointsTo}(\text{?from}, \text{?heap}) \)

never iterate over full views

- \( \Delta \text{Assign}(\text{?to}, \text{?from}), \text{VarPointsTo}(\text{?from}, \text{?heap}) \)
- \( \Delta \text{Assign}(\text{?to}, \text{?from}), \text{VarPointsTo}(\text{?heap}, \text{?from}) \)
optimization principles

always use index efficiently

- \text{Assign}(\text{?from}, \text{?to}), \Delta\text{VarPointsTo}(\text{?from}, \text{?heap})
- \text{Assign}(\text{?to}, \text{?from}), \Delta\text{VarPointsTo}(\text{?from}, \text{?heap})

never iterate over full views

- \Delta\text{Assign}(\text{?to}, \text{?from}), \text{VarPointsTo}(\text{?from}, \text{?heap})
- \Delta\text{Assign}(\text{?to}, \text{?from}), \text{VarPointsTo}(\text{?heap}, \text{?from})

never iterate over big input relations

- \text{StoreInstanceField}(\text{?from}, \text{?signature}, \text{?base}), \text{VarPointsTo}(\text{?baseheap}, \text{?base}), \Delta\text{VarPointsTo}(\text{?heap}, \text{?from})
- we’ll get back to that...
always use index efficiently

- Assign(?from, ?to), \( \Delta \text{VarPointsTo}(?from, ?heap) \)
- Assign(?to, ?from), \( \Delta \text{VarPointsTo}(?from, ?heap) \)

never iterate over full views

- \( \Delta \text{Assign}(?to, ?from) \), VarPointsTo(?from, ?heap)
- \( \Delta \text{Assign}(?to, ?from) \), VarPointsTo(?heap, ?from)

never iterate over big input relations

- StoreInstanceField(?from, ?signature, ?base), VarPointsTo(?baseheap, ?base), \( \Delta \text{VarPointsTo}(?heap, ?from) \)

iterate over delta
access big relations with index

we'll get back to that...
unoptimized

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

constraints

<table>
<thead>
<tr>
<th></th>
<th>Assign</th>
<th>VarPointsTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ΔAssign</td>
<td>-</td>
<td>?from</td>
</tr>
<tr>
<td>2. VarPointsTo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. ΔVarPointsTo</td>
<td>?from</td>
<td>-</td>
</tr>
<tr>
<td>2. Assign</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

optimized

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

\[
\text{VarPointsTo}(\text{?to}, \text{?heap}) \leftarrow \\
\text{Assign}(\text{?from}, \text{?to}), \\
\text{VarPointsTo}(\text{?from}, \text{?heap}).
\]

constraints

<table>
<thead>
<tr>
<th>1. (\Delta\text{Assign} )</th>
<th>2. (\Delta\text{VarPointsTo} )</th>
<th>Assign</th>
<th>(\text{VarPointsTo} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{?from} )</td>
<td>(\text{?from} )</td>
<td>-</td>
<td>(\text{?from} )</td>
</tr>
<tr>
<td>(\text{?from} )</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

optimized

\[
\text{VarPointsTo}(\text{?heap}, \text{?to}) \leftarrow \\
\text{Assign}(\text{?to}, \text{?from}), \\
\text{VarPointsTo}(\text{?heap}, \text{?from}).
\]

other join orderings not interesting: pruned immediately
InstanceFieldPointsTo(\texttt{?baseheap}, \texttt{?signature}, \texttt{?heap}) \leftarrow\ 
\text{StoreInstanceField}(\texttt{?from}, \texttt{?base}, \texttt{?signature}), \ 
\text{VarPointsTo}_{1}(\texttt{?baseheap}, \texttt{?base}), \ 
\text{VarPointsTo}_{2}(\texttt{?heap}, \texttt{?from}).

<table>
<thead>
<tr>
<th>1. $\Delta$VarPointsTo$_{1}$</th>
<th>Store</th>
<th>VarPointsTo$_{1}$</th>
<th>VarPointsTo$_{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. StoreInstanceField</td>
<td>$\texttt{?base}$</td>
<td>$-$</td>
<td>$\texttt{?from}$</td>
</tr>
<tr>
<td>3. VarPointsTo$_{2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1. $\Delta$VarPointsTo$_{2}$</th>
<th>Store</th>
<th>VarPointsTo$_{1}$</th>
<th>VarPointsTo$_{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. StoreInstanceField</td>
<td>$\texttt{?from}$</td>
<td>$\texttt{?base}$</td>
<td>$-$</td>
</tr>
<tr>
<td>3. VarPointsTo$_{1}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <-
StoreInstanceField(?from, ?base, ?signature),
VarPointsTo_1(?baseheap, ?base),
VarPointsTo_2(?heap, ?from).

<table>
<thead>
<tr>
<th>1. ΔVarPointsTo_1</th>
<th>Store</th>
<th>VarPointsTo_1</th>
<th>VarPointsTo_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. StoreInstanceField</td>
<td>?base</td>
<td>-</td>
<td>?from</td>
</tr>
<tr>
<td>3. VarPointsTo_2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1. ΔVarPointsTo_2</th>
<th>Store</th>
<th>VarPointsTo_1</th>
<th>VarPointsTo_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. StoreInstanceField</td>
<td>?from</td>
<td>?base</td>
<td>-</td>
</tr>
<tr>
<td>3. VarPointsTo_1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

conflicting constraints! there is no efficient index.
optimization: folding

\begin{verbatim}
InstanceFieldPointsTo(?baseheap, ?signature, ?heap) <- StoreInstanceField(?from, ?base, ?signature), VarPointsTo_1(?baseheap, ?base), VarPointsTo_2(?heap, ?from).

fold StoreInstanceField and VarPointsTo_1 to create an index:

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


we have just introduced a materialized view!
\end{verbatim}
method invocations: propagated exceptions

\[\text{ThrowPointsTo}(\text{?heap, ?callerMethod}) \leftarrow \]
\[\quad \text{CallGraphEdge}(\text{?invocation, ?tomethod}),\]
\[\quad \text{ThrowPointsTo}(\text{?heap, ?tomethod}),\]
\[\quad \text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype},\]
\[\quad \text{not exists ExceptionHandler}[\text{?heaptype, ?invocation}],\]
\[\quad \text{Instruction:Method}[\text{?invocation}] = \text{?callerMethod}.\]

method invocations: caught exceptions

\[\text{VarPointsTo}(\text{?heap, ?param}) \leftarrow \]
\[\quad \text{CallGraphEdge}(\text{?invocation, ?tomethod}),\]
\[\quad \text{ThrowPointsTo}(\text{?heap, ?tomethod}),\]
\[\quad \text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype},\]
\[\quad \text{ExceptionHandler}[\text{?heaptype, ?invocation}] = \text{?handler},\]
\[\quad \text{ExceptionHandler:FormalParam}[\text{?handler}] = \text{?param}.\]
precise exception analysis

method invocations: propagated exceptions

\[\text{ThrowPointsTo}(\text{heap}, \text{callerMethod}) \leftarrow \]
\[\text{CallGraphEdge}(\text{invocation}, \text{tomethod}),\]
\[\text{ThrowPointsTo}(\text{heap}, \text{tomethod}),\]
\[\text{HeapAllocation}:\text{Type}[\text{heap}] = \text{heaptype},\]
\[\text{not exists ExceptionHandler}[\text{heaptype}, \text{invocation}],\]
\[\text{Instruction}:\text{Method}[\text{invocation}] = \text{callerMethod}.\]

method invocations: caught exceptions

\[\text{VarPointsTo}(\text{heap}, \text{param}) \leftarrow \]
\[\text{CallGraphEdge}(\text{invocation}, \text{tomethod}),\]
\[\text{ThrowPointsTo}(\text{heap}, \text{tomethod}),\]
\[\text{HeapAllocation}:\text{Type}[\text{heap}] = \text{heaptype},\]
\[\text{ExceptionHandler}[\text{heaptype}, \text{invocation}] = \text{handler},\]
\[\text{ExceptionHandler}:\text{FormalParam}[\text{handler}] = \text{param}.\]
precise exception analysis

**method invocations: propagated exceptions**

\[
\text{ThrowPointsTo}(\text{heap}, \text{callerMethod}) \leftarrow \text{CallGraphEdge}(\text{invocation}, \text{tomethod}), \text{ThrowPointsTo}(\text{heap}, \text{tomethod}), \text{HeapAllocation:Type}[\text{heap}] = \text{heaptype}, \text{not exists ExceptionHandler}[\text{heaptype}, \text{invocation}], \text{Instruction:Method}[\text{invocation}] = \text{callerMethod}.
\]

**method invocations: caught exceptions**

\[
\text{VarPointsTo}(\text{heap}, \text{param}) \leftarrow \text{CallGraphEdge}(\text{invocation}, \text{tomethod}), \text{ThrowPointsTo}(\text{heap}, \text{tomethod}), \text{HeapAllocation:Type}[\text{heap}] = \text{heaptype}, \text{ExceptionHandler}[\text{heaptype}, \text{invocation}] = \text{handler}, \text{ExceptionHandler:FormalParam}[\text{handler}] = \text{param}.
\]
precise exception analysis

method invocations: propagated exceptions

\[
\text{ThrowPointsTo}(\text{?heap}, \text{?callerMethod}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype}, \\
\text{not exists ExceptionHandler}[\text{?heaptype}, \text{?invocation}], \\
\text{Instruction:Method}[\text{?invocation}] = \text{?callerMethod}.
\]

method invocations: caught exceptions

\[
\text{VarPointsTo}(\text{?heap}, \text{?param}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype}, \\
\text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}] = \text{?handler}, \\
\text{ExceptionHandler:FormalParam}[\text{?handler}] = \text{?param}.
\]
precise exception analysis

**method invocations: propagated exceptions**

\[
\text{ThrowPointsTo}(\text{?heap}, \text{?callerMethod}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation}:\text{Type}[\text{?heap}] = \text{?heaptype}, \\
\text{not exists } \text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}], \\
\text{Instruction}:\text{Method}[\text{?invocation}] = \text{?callerMethod}.
\]

**method invocations: caught exceptions**

\[
\text{VarPointsTo}(\text{?heap}, \text{?param}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation}:\text{Type}[\text{?heap}] = \text{?heaptype}, \\
\text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}] = \text{?handler}, \\
\text{ExceptionHandler}:\text{FormalParam}[\text{?handler}] = \text{?param}.
\]
precise exception analysis

**method invocations: propagated exceptions**

\[
\text{ThrowPointsTo}(\text{?heap}, \text{?callerMethod}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype}, \\
\text{not exists ExceptionHandler}[\text{?heaptype}, \text{?invocation}], \\
\text{Instruction:Method}[\text{?invocation}] = \text{?callerMethod}.
\]

**method invocations: caught exceptions**

\[
\text{VarPointsTo}(\text{?heap}, \text{?param}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype}, \\
\text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}] = \text{?handler}, \\
\text{ExceptionHandler:FormalParam}[\text{?handler}] = \text{?param}.
\]
method invocations: propagated exceptions

\[
\text{ThrowPointsTo}(\text{?heap}, \text{?callerMethod}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype}, \\
\text{not exists ExceptionHandler}[\text{?heaptype}, \text{?invocation}], \\
\text{Instruction:Method}[\text{?invocation}] = \text{?callerMethod}.
\]

method invocations: caught exceptions

\[
\text{VarPointsTo}(\text{?heap}, \text{?param}) \leftarrow \\
\text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\text{HeapAllocation:Type}[\text{?heap}] = \text{?heaptype}, \\
\text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}] = \text{?handler}, \\
\text{ExceptionHandler:FormalParam}[\text{?handler}] = \text{?param}.
\]
precise exception analysis

method invocations: propagated exceptions

\[
\text{ThrowPointsTo}(\text{?heap}, \text{?callerMethod}) \leftarrow \\
\quad \text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\quad \text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\quad \text{HeapAllocation}: \text{Type}[\text{?heap}] = \text{?heaptype}, \\
\quad \text{not exists } \text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}], \\
\quad \text{Instruction}: \text{Method}[\text{?invocation}] = \text{?callerMethod}.
\]

method invocations: caught exceptions

\[
\text{VarPointsTo}(\text{?heap}, \text{?param}) \leftarrow \\
\quad \text{CallGraphEdge}(\text{?invocation}, \text{?tomethod}), \\
\quad \text{ThrowPointsTo}(\text{?heap}, \text{?tomethod}), \\
\quad \text{HeapAllocation}: \text{Type}[\text{?heap}] = \text{?heaptype}, \\
\quad \text{ExceptionHandler}[\text{?heaptype}, \text{?invocation}] = \text{?handler}, \\
\quad \text{ExceptionHandler}: \text{FormalParam}[\text{?handler}] = \text{?param}.
\]
precise exception analysis

method invocations: propagated exceptions

\[\text{ThrowPointsTo}(?\text{heap}, \text{?callerMethod}) \leftarrow \]
\[\text{CallGraphEdge}(?\text{invocation}, \text{?tomethod}),\]
\[\text{ThrowPointsTo}(?\text{heap}, \text{?tomethod}),\]
\[\text{HeapAllocation}:\text{Type}[?\text{heap}] = ?\text{heaptype},\]
\[\text{not exists ExceptionHandler}[?\text{heaptype}, ?\text{invocation}],\]
\[\text{Instruction}:\text{Method}[?\text{invocation}] = ?\text{callerMethod}.\]

method invocations: caught exceptions

\[\text{VarPointsTo}(?\text{heap}, \text{?param}) \leftarrow \]
\[\text{CallGraphEdge}(?\text{invocation}, \text{?tomethod}),\]
\[\text{ThrowPointsTo}(?\text{heap}, \text{?tomethod}),\]
\[\text{HeapAllocation}:\text{Type}[?\text{heap}] = ?\text{heaptype},\]
\[\text{ExceptionHandler}[?\text{heaptype}, ?\text{invocation}] = ?\text{handler},\]
\[\text{ExceptionHandler}:\text{FormalParam}[?\text{handler}] = ?\text{param}.\]
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