Strictly Declarative Points-to Analysis

Martin Bravenboer

University of Massachusetts Amherst
University of Oregon

March 30, 2009

New York University

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

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

overview 1

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

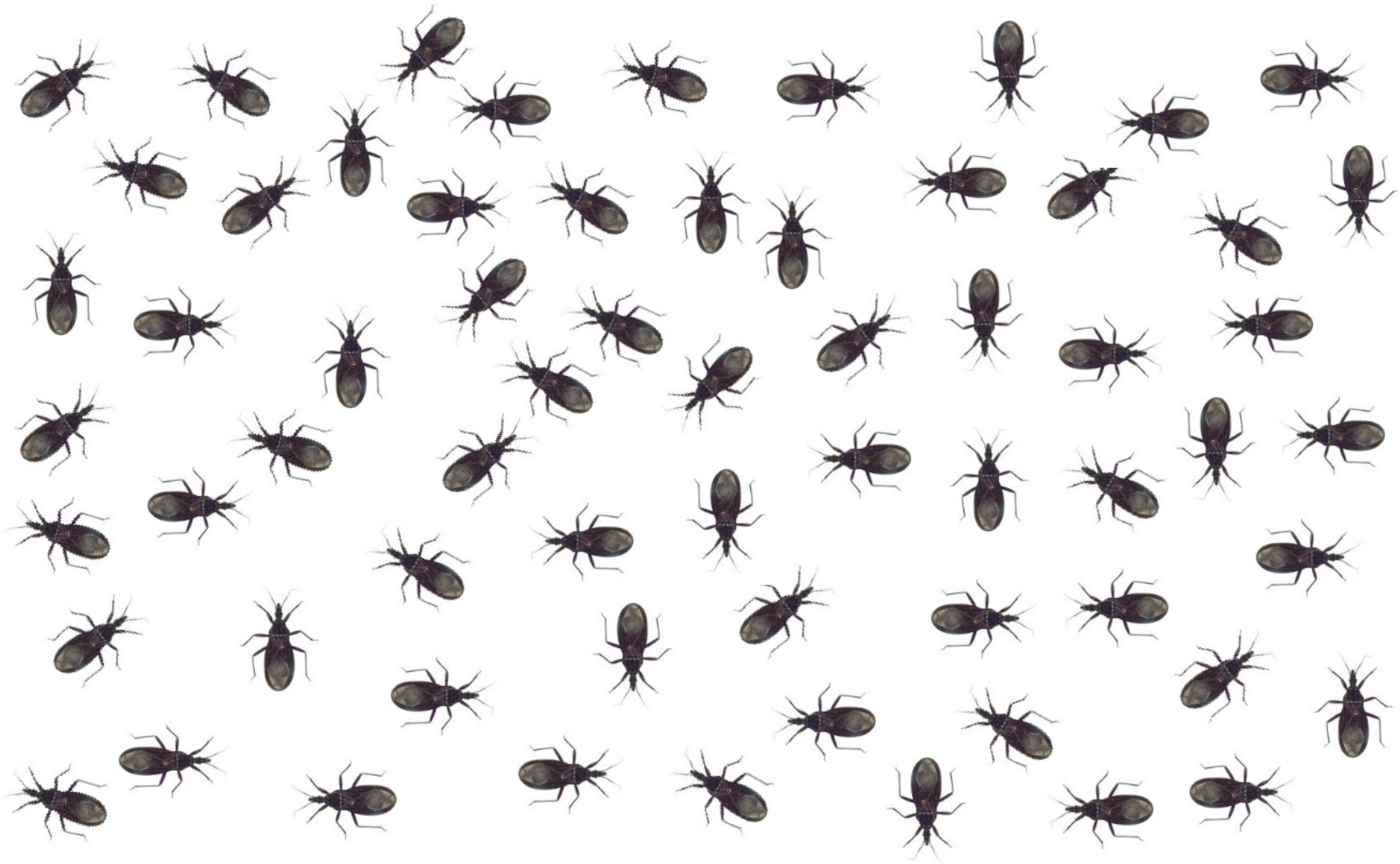
why is it relevant?

- optimization
- understanding, find bugs

program analysis: run faster


NO
UNDER-
STANDING
ANY
TIME

program analysis: find bugs



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()

objects represented by allocation sites

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()

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()

necessarily an approximation

full precision too expensive

pointer analysis: a complex domain

Results 1 - 20 of 2,343

Save results to a Binder

Sort by relevance in expanded form

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

pointer analysis: a complex domain

flow-sensitive

Results 1 - 20 of 2,343

Save results to a Binder

Sort by relevance in expanded form

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

pointer analysis: a complex domain

flow-sensitive

field-sensitive

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

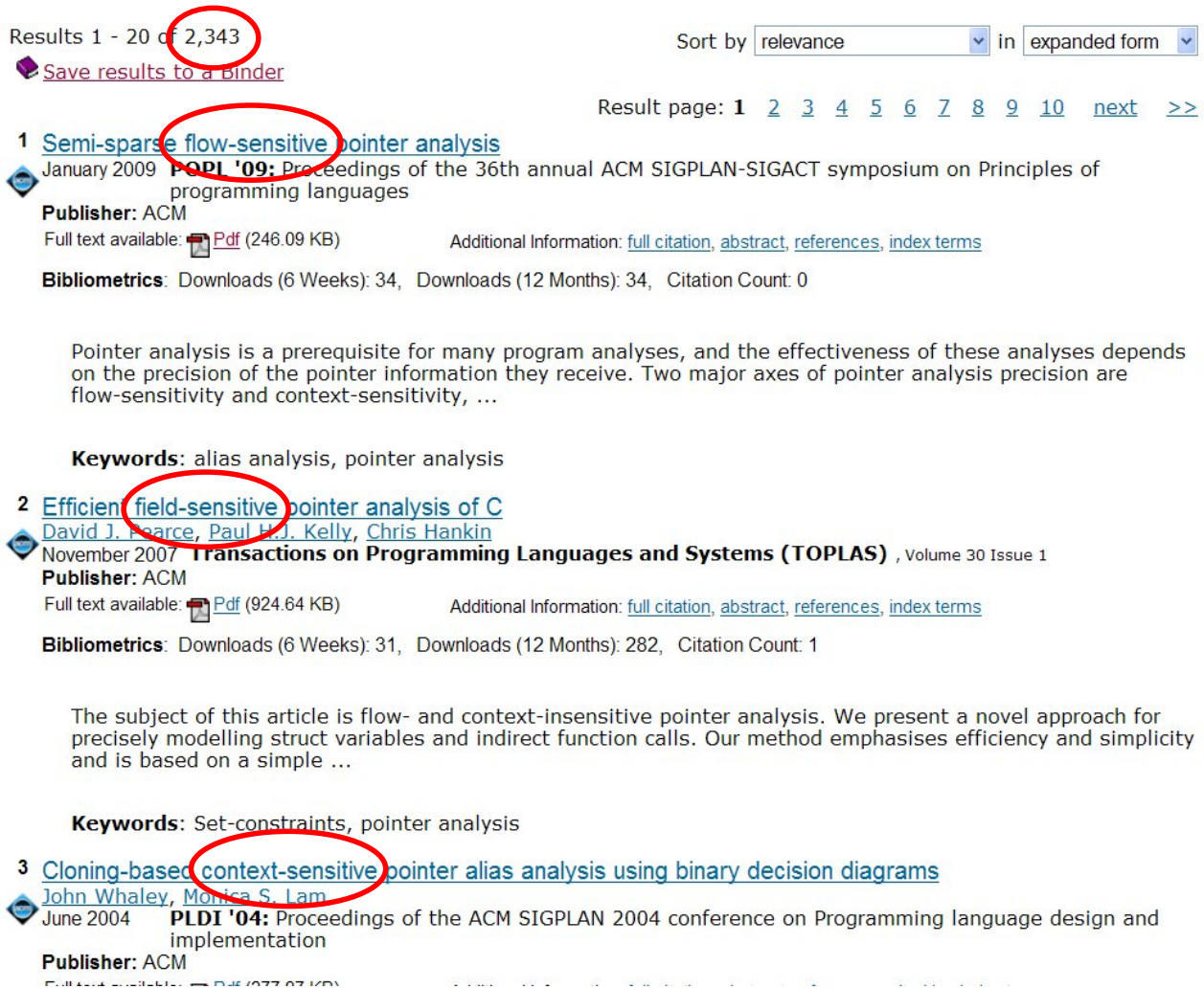
Publisher: ACM

pointer analysis: a complex domain

flow-sensitive

context-sensitive

field-sensitive



Results 1 - 20 of 2,343

Save results to a Binder

Sort by relevance in expanded form

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

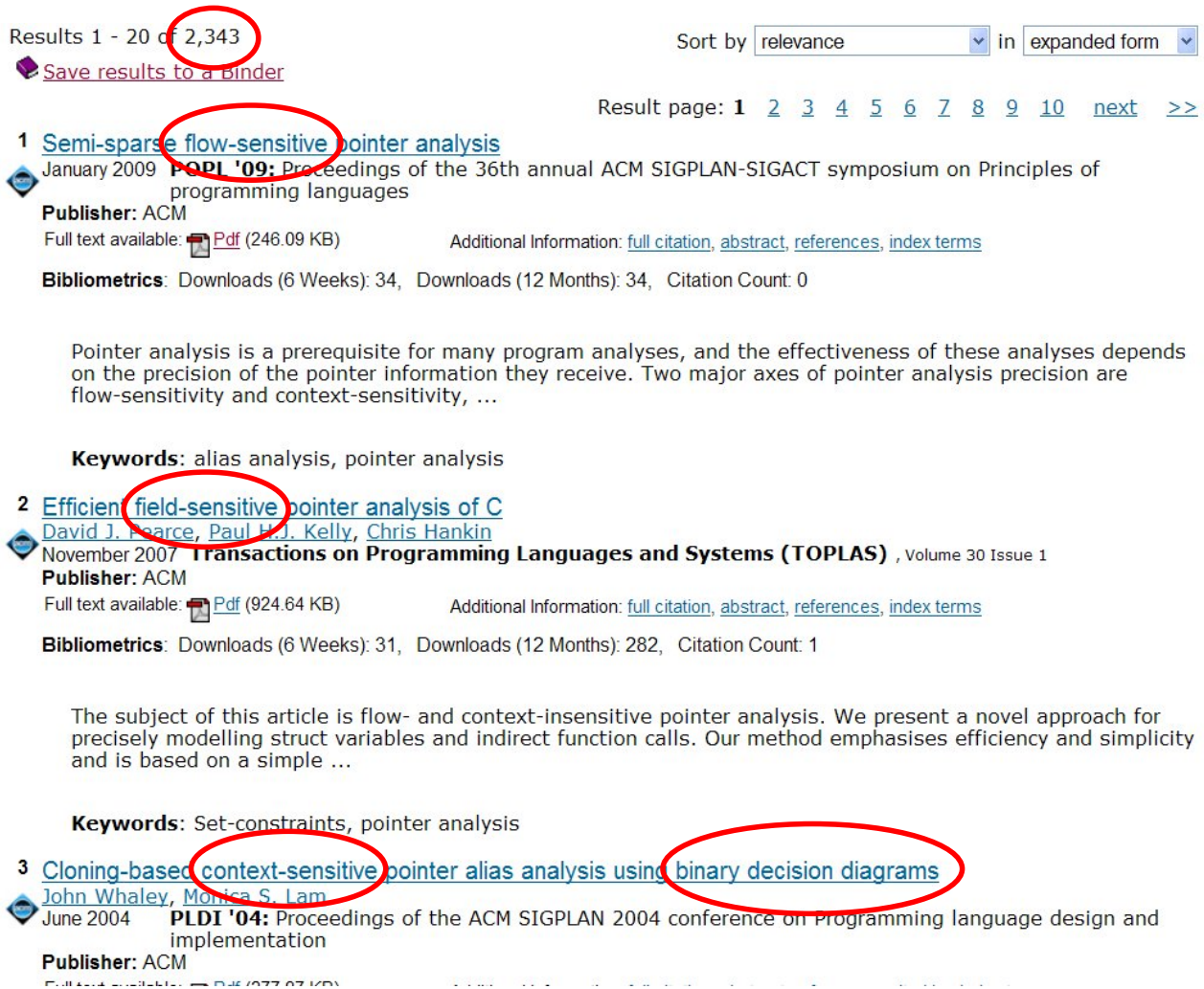
pointer analysis: a complex domain

flow-sensitive

context-sensitive

field-sensitive

binary decision diagrams



Results 1 - 20 of 2,343

Save results to a Binder

Sort by relevance in expanded form

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

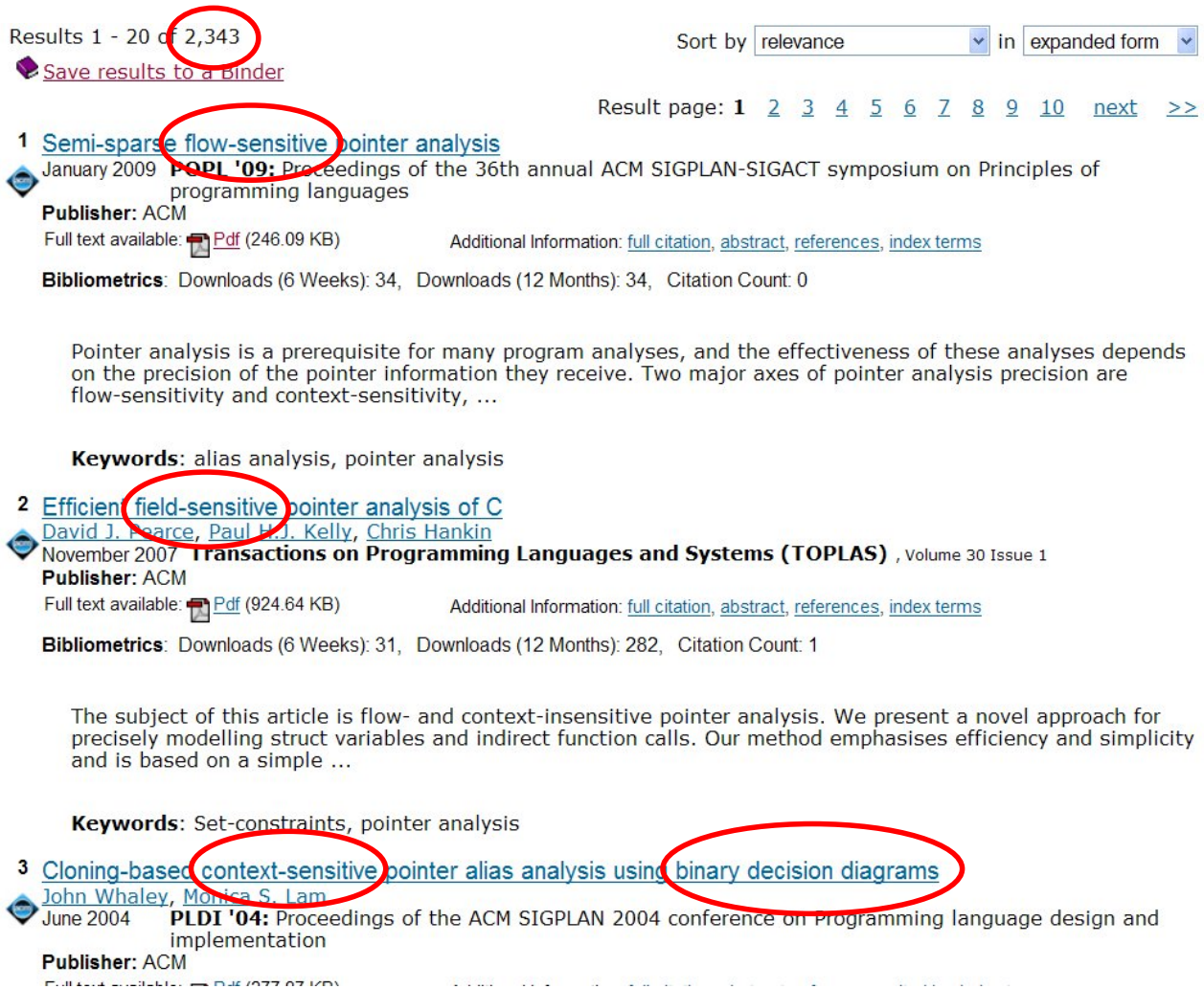
John Whaley, Monica S. Lam

June 2004 **PLDI '04:** Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation

Publisher: ACM

pointer analysis: a complex domain

- inclusion-based
- unification-based
- flow-sensitive
- on-the-fly call graph
- context-sensitive
- k-cfa
- object sensitive
- field-based
- field-sensitive
- heap cloning
- binary decision diagrams
- 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 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: a
begin
1. i
2. f
3. v
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

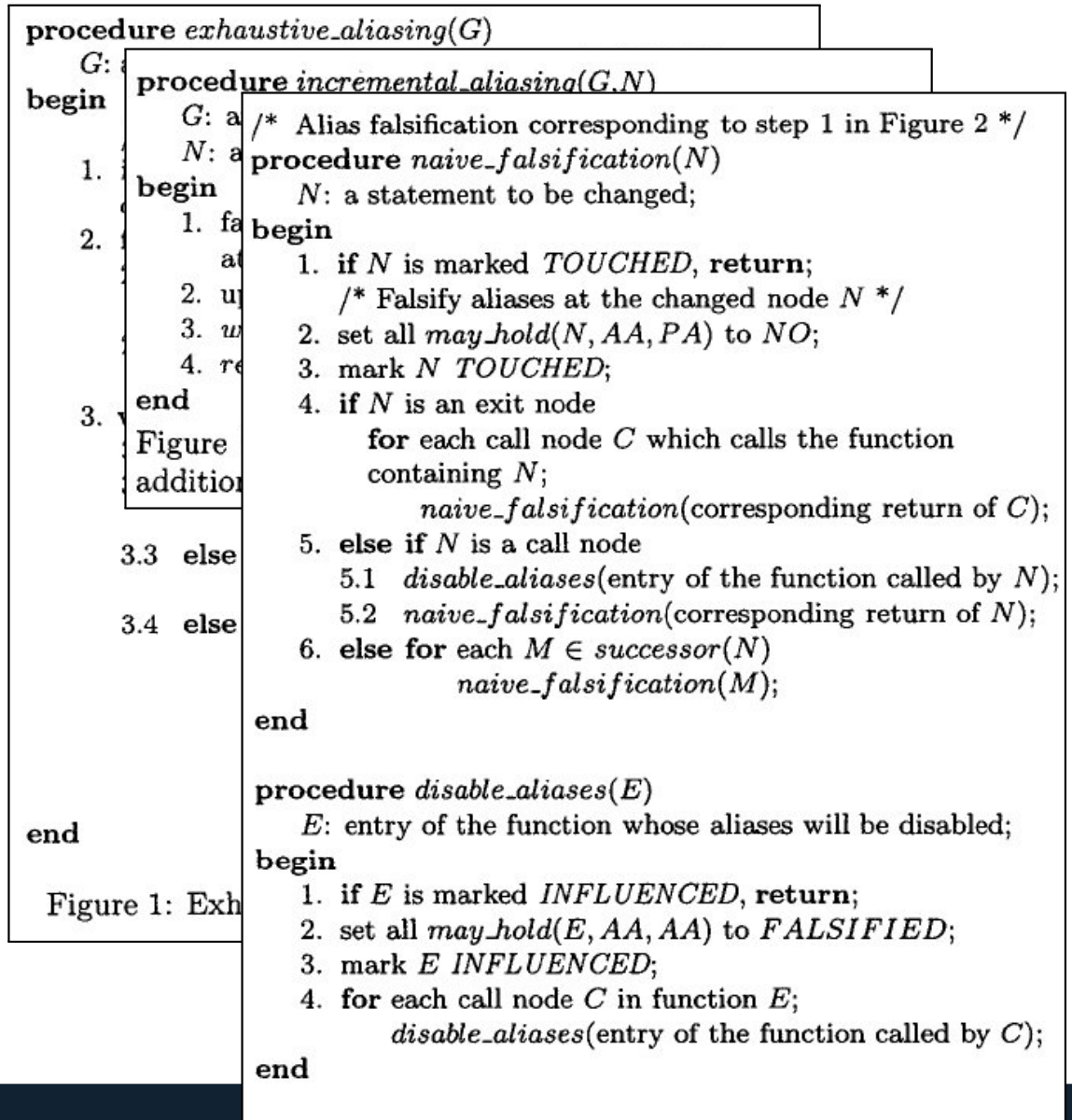
Figure 1: Exhaustive algorithm for *pointer aliasing*

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

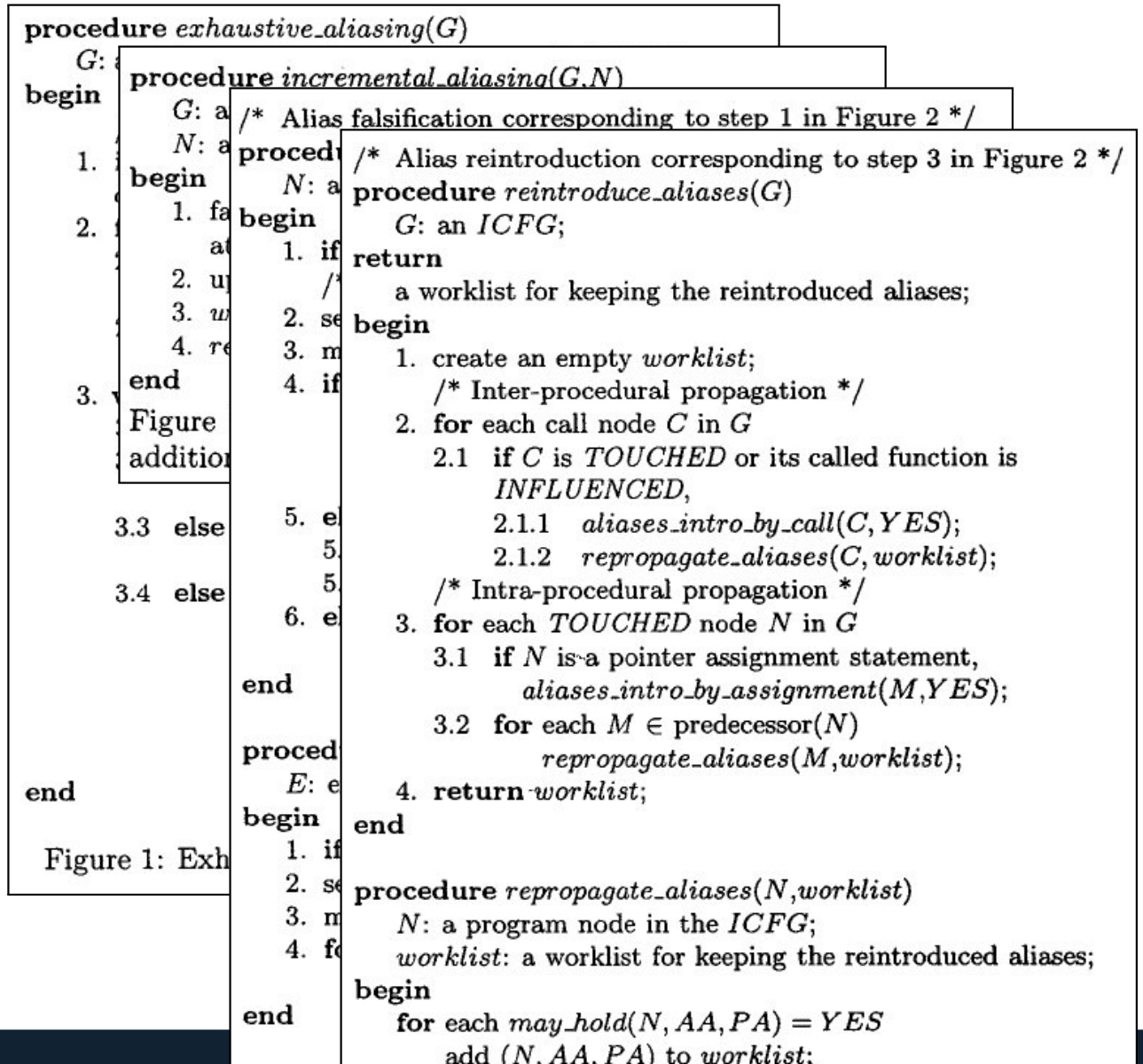
algorithms in a 10-page pointer analysis paper



```
/* 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 ∈ 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
```

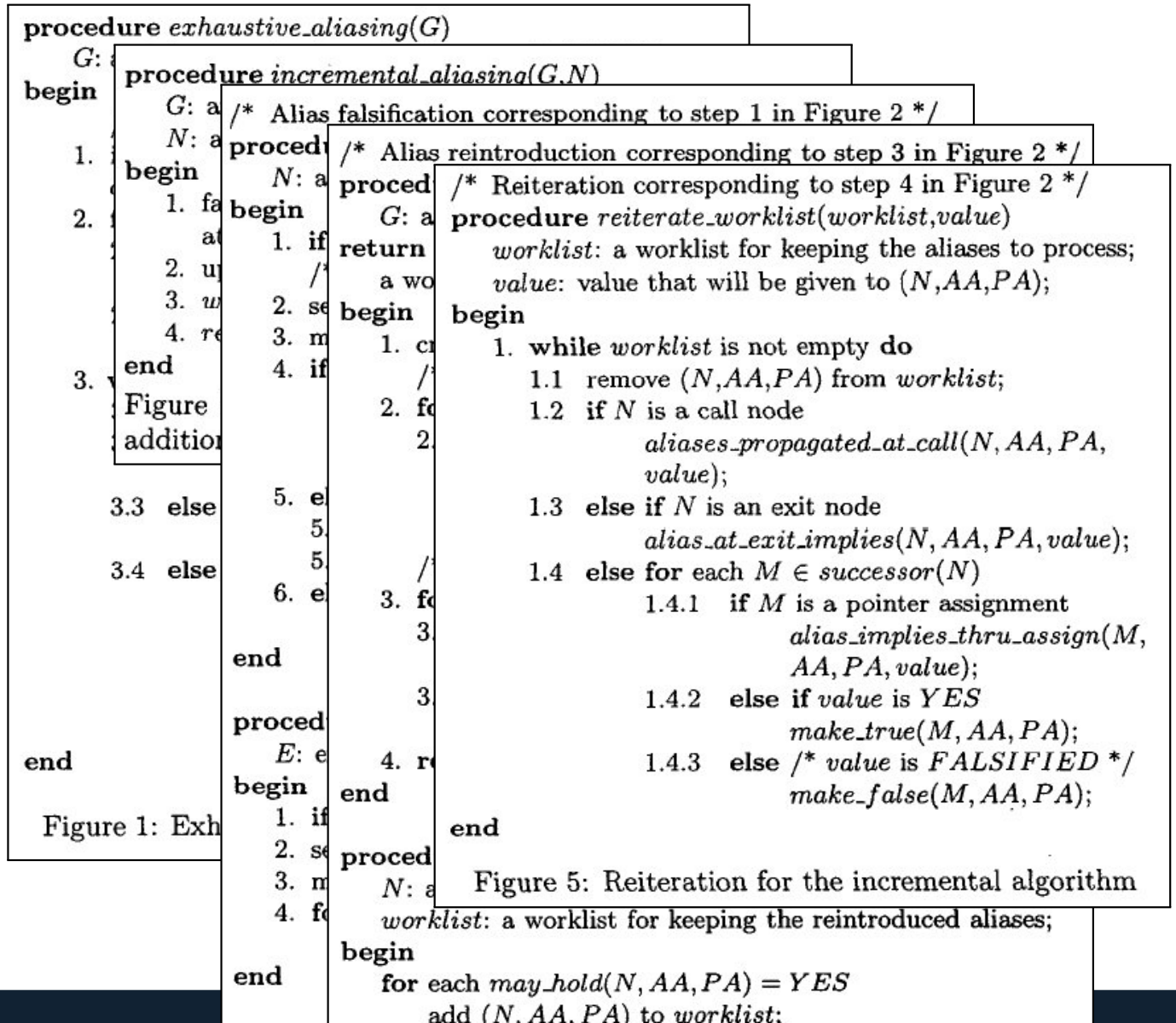
algorithms in a 10-page pointer analysis paper 7



```
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 ∈ 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;
```

algorithms in a 10-page pointer analysis paper



```
procedure exhaustive_aliasing(G)
    G: a
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. w
    4. re
end
```

Figure
additio

```
/* Alias falsification corresponding to step 1 in Figure 2 */
proced
    N: a
begin
    1. if
       /*
    2. se
    3. m
    4. if
    5. e
       5.
       5.
       /*
    6. e
end

proced
    E: e
begin
    1. if
    2. se
    3. m
    4. fo
end
```

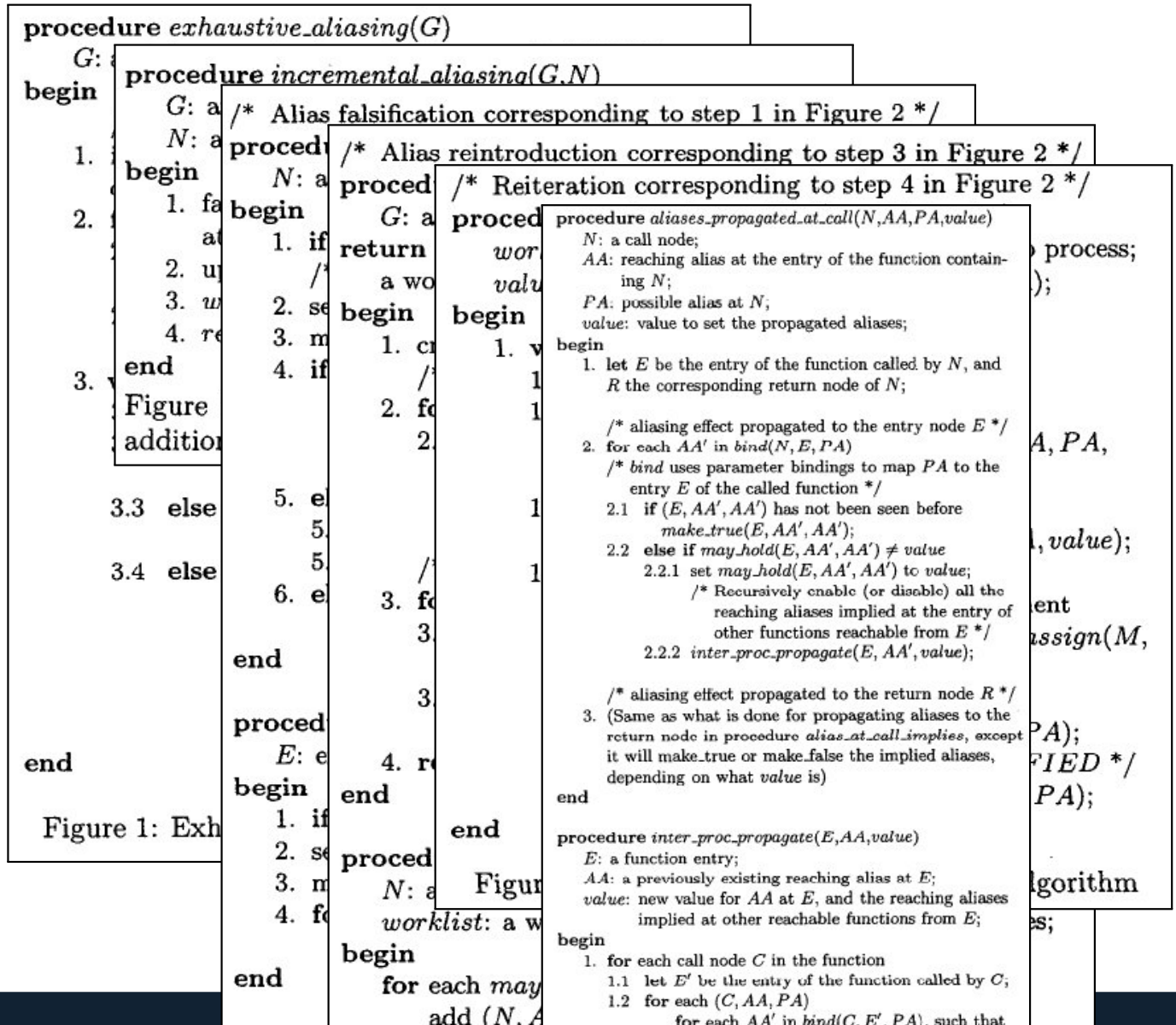
```
/* Alias reintroduction corresponding to step 3 in Figure 2 */
proced
    G: a
return
    a wo
begin
    1. cr
       /*
    2. fo
       2.
       /*
    3. fo
       3.
       3.
    4. re
end

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

```
/* 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 ∈ 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

algorithms in a 10-page pointer analysis paper



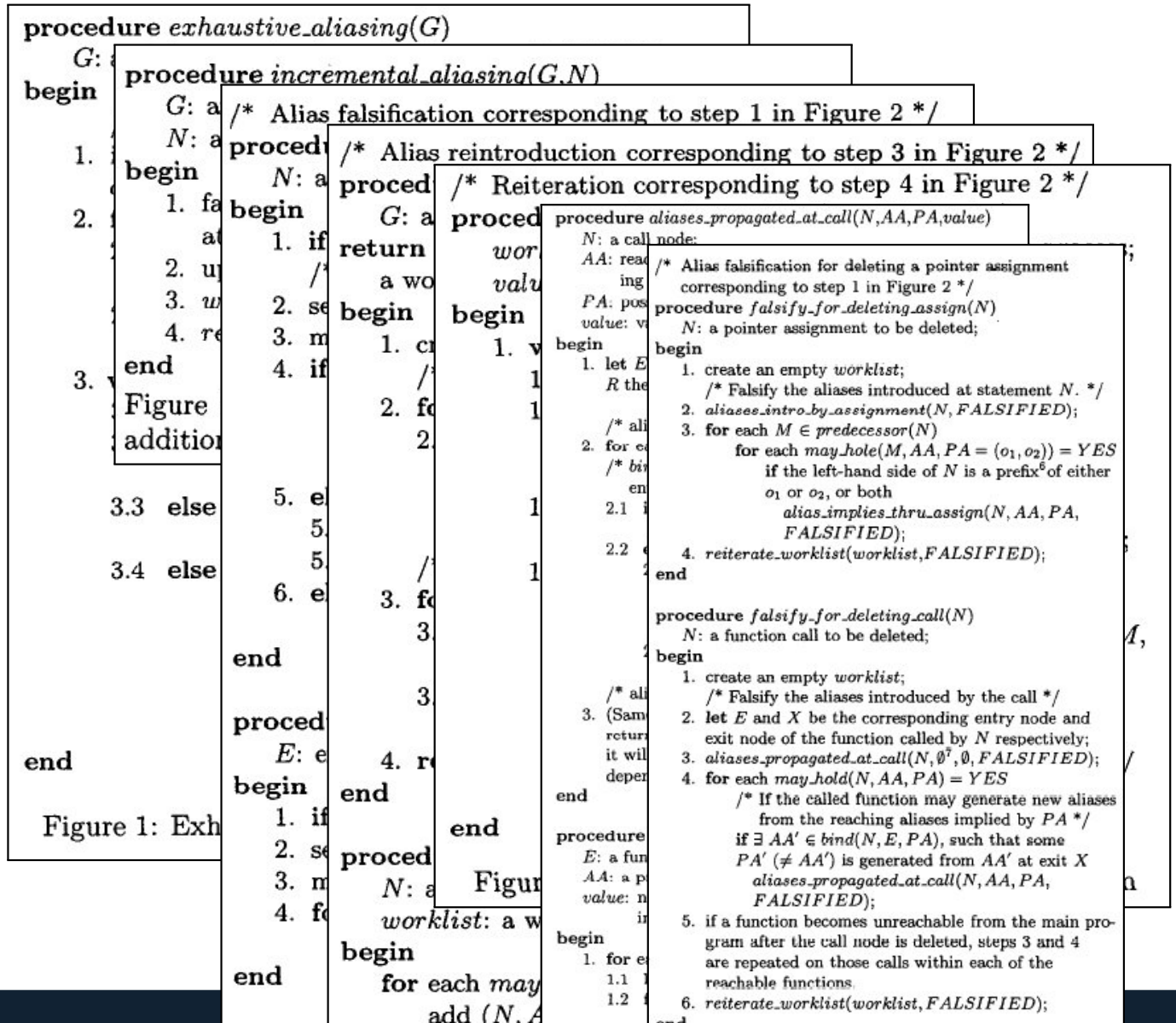
procedure *aliases_propagated_at_call*(*N*,*AA*,*PA*,*value*)
N: a call node;
AA: reaching alias at the entry of the function containing *N*;
PA: possible alias at *N*;
value: value to set the propagated aliases;
begin
1. let *E* be the entry of the function called by *N*, and *R* the corresponding return 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 $may\_hold(E, AA', AA') \neq value$
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)
for each AA' in $bind(C, E', PA)$ such that

algorithms in a 10-page pointer analysis paper



```
procedure exhaustive_aliasing(G)
    G: a
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. re
end
```

Figure
additio

```
/* Alias falsification corresponding to step 1 in Figure 2 */
proced
    N: a
begin
    1. if
       /*
    2. se
    3. m
    4. if
    5. e
       5.
       5.
    6. e
end

proced
    E: e
begin
    1. if
    2. se
    3. m
    4. fo
end
```

```
/* Alias reintroduction corresponding to step 3 in Figure 2 */
proced
    G: a
return
    a wo
begin
    1. cr
       /*
    2. fo
       2.
       3.
    3. fo
       3.
       3.
    4. re
end

proced
    N: a
    worklist: a w
begin
    for each may
        add (N, A
```

```
/* Reiteration corresponding to step 4 in Figure 2 */
proced
    wor
    valu
begin
    1. v
       1
       1
       1
       1
end

Figur
```

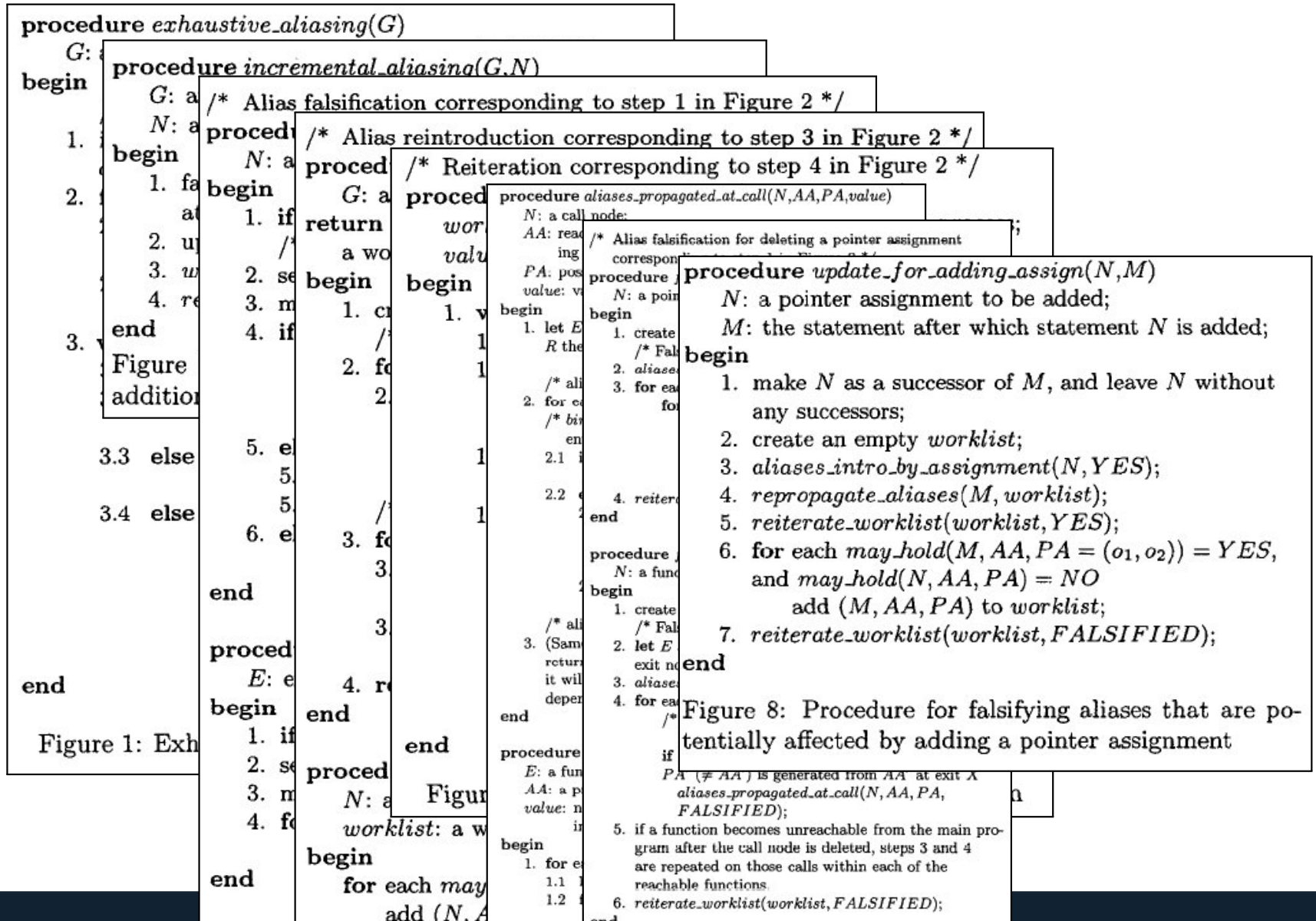
```
procedure aliases_propagated_at_call(N,AA,PA,value)
    N: a call node;
    AA: rea
        ing
    PA: pos
    value: v
begin
    1. let E
       R the
       /* ali
    2. for e
       /* bi
          en
       2.1 i
       2.2
       /* ali
    3. (Sam
       retur
       it wil
       deper
end

procedure
    E: a fun
    AA: a p
    value: n
        i
begin
    1. for e
       1.1
       1.2
```

```
/* Alias falsification for deleting a pointer assignment
   corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_assign(N)
    N: a pointer assignment to be deleted;
begin
    1. create an empty worklist;
       /* Falsify the aliases introduced at statement N. */
    2. aliases_intro_by_assignment(N, FALSIFIED);
    3. for each M ∈ predecessor(N)
           for each may_hole(M, AA, PA = (o1, o2)) = YES
               if the left-hand side of N is a prefix[6] of either
               o1 or o2, or both
                   alias_implies_thru_assign(N, AA, PA,
                   FALSIFIED);
    4. reiterate_worklist(worklist,FALSIFIED);
end

procedure falsify_for_deleting_call(N)
    N: a function call to be deleted;
begin
    1. create an empty worklist;
       /* Falsify the aliases introduced by the call */
    2. let E and X be the corresponding entry node and
       exit node of the function called by N respectively;
    3. aliases_propagated_at_call(N, ∅², ∅, FALSIFIED);
    4. for each may_hold(N, AA, PA) = YES
           /* If the called function may generate new aliases
              from the reaching aliases implied by PA */
           if ∃ AA' ∈ bind(N, E, PA), such that some
           PA' (≠ AA') is generated from AA' at exit X
               aliases_propagated_at_call(N, AA, PA,
               FALSIFIED);
    5. if a function becomes unreachable from the main pro-
       gram 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);
end
```

algorithms in a 10-page pointer analysis paper



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*$(M, AA, PA = (o_1, o_2)) = YES$, 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

algorithms in a 10-page pointer analysis paper

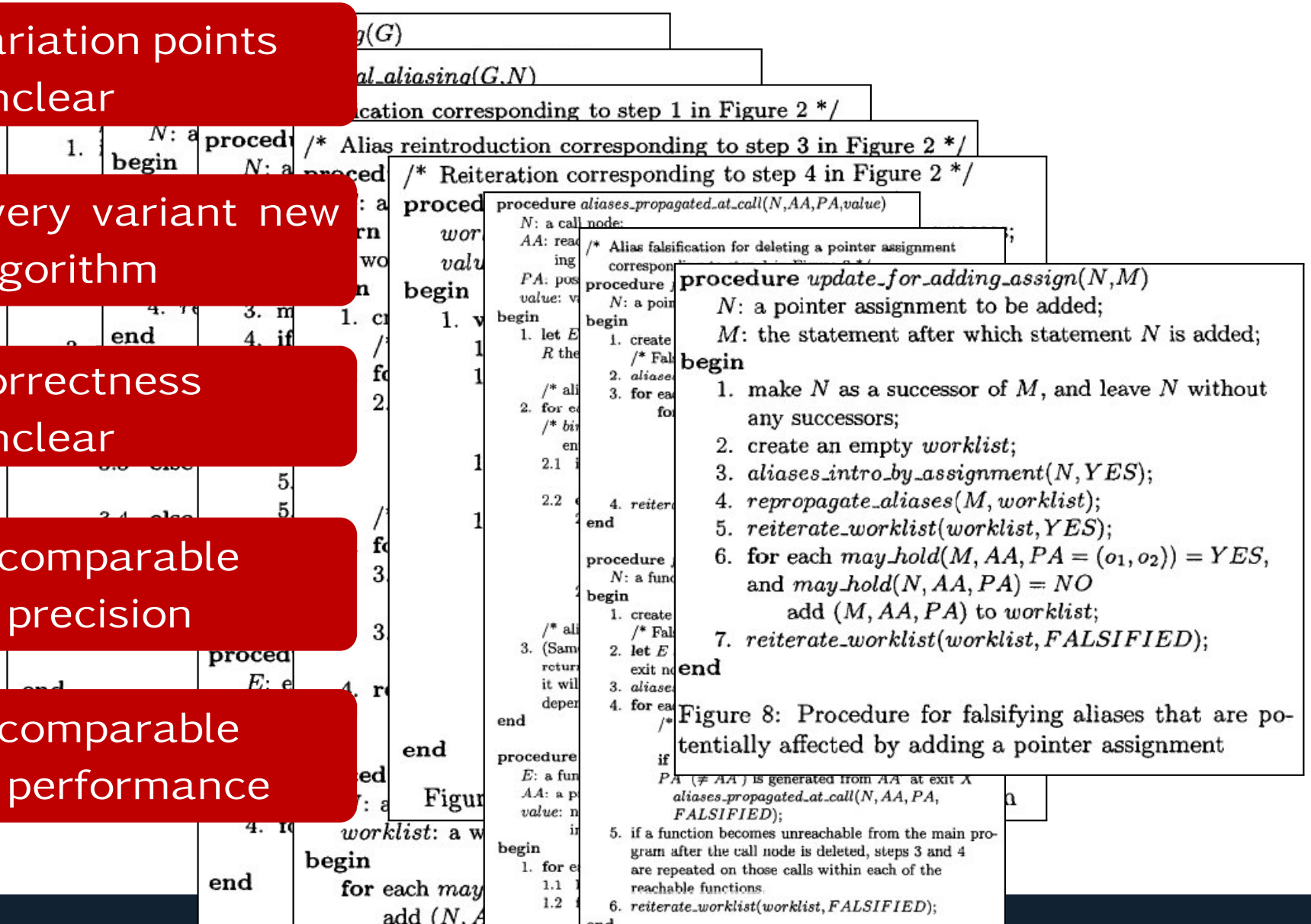
variation points unclear

every variant new algorithm

correctness unclear

incomparable in precision

incomparable in performance

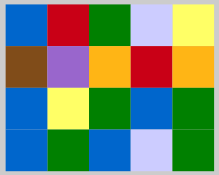


procedure *update_for_adding_assign*(N,M)
N: a pointer assignment to be added;
M: the statement after which statement N is added;
begin
1. make N as a successor of M, and leave N without any successors;
2. create an empty *worklist*;
3. *aliases_intro_by_assignment*(N, YES);
4. *repropagate_aliases*$(M, worklist)$;
5. *reiterate_worklist*$(worklist, YES)$;
6. **for** each $may\_hold(M, AA, PA = (o_1, o_2)) = YES$, and $may\_hold(N, AA, PA) = NO$
   add (M, AA, PA) to *worklist*;
7. *reiterate_worklist*$(worklist, FALSIFIED)$;

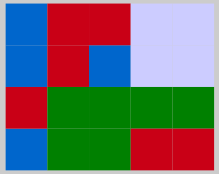
end

Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment

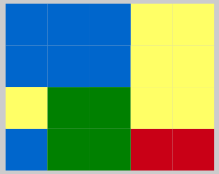
algorithm



algorithm

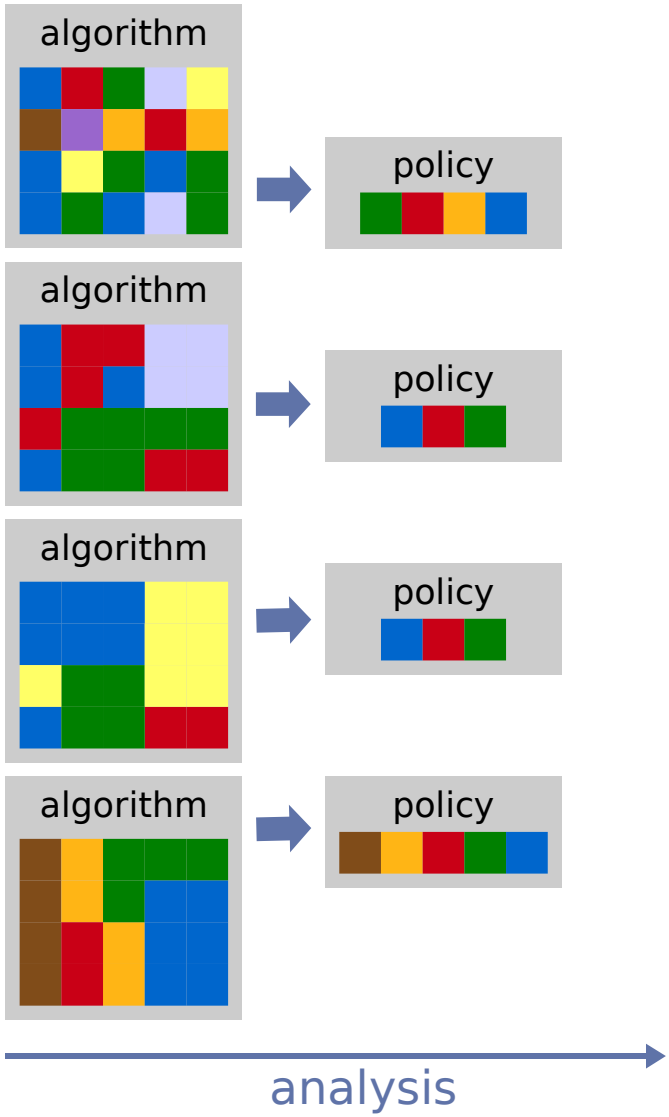


algorithm

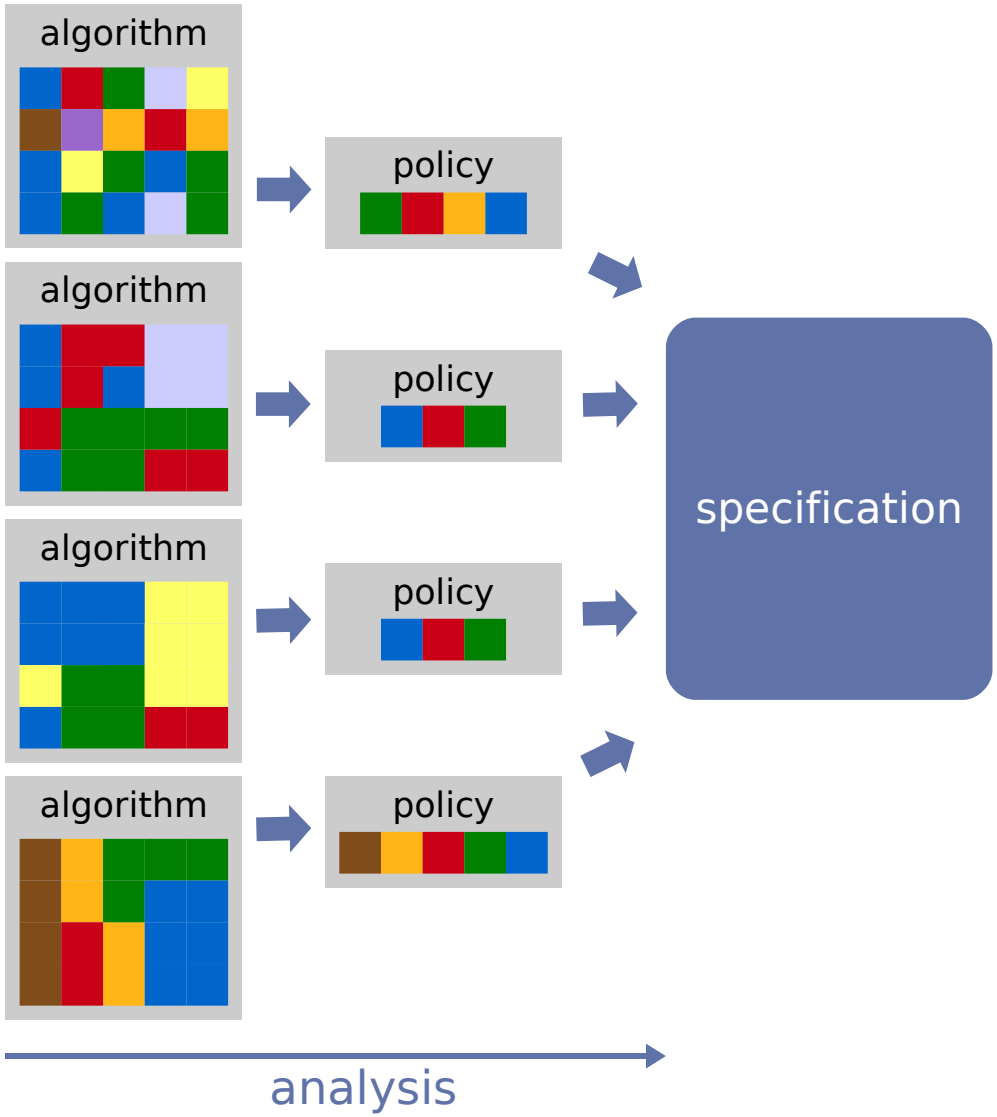


algorithm

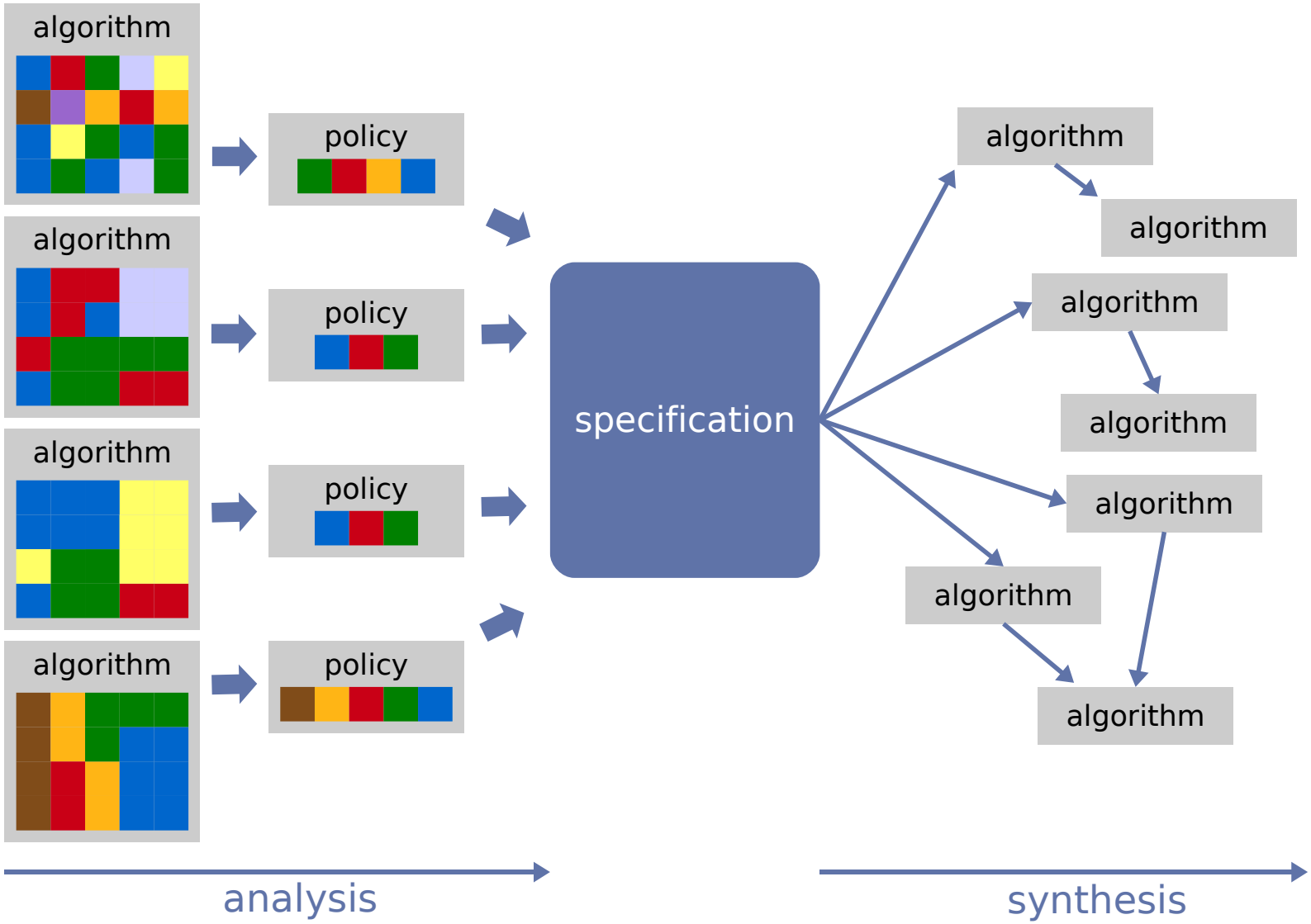
from algorithms to specification and back



from algorithms to specification and back



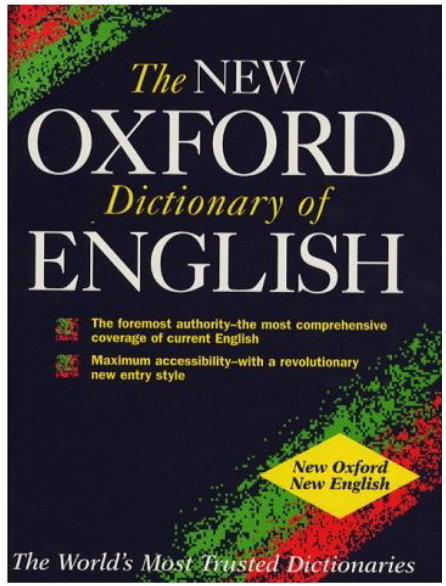
from algorithms to specification and back



from algorithms to specification

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

context-sensitive pointer analysis for java

paddle

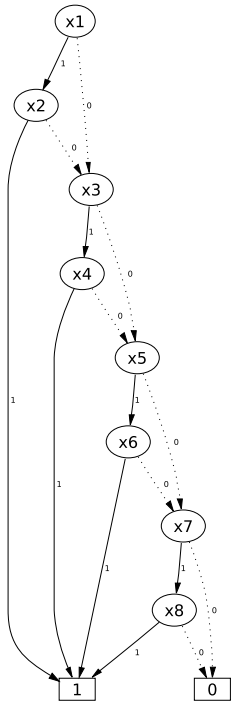
java + relational algebra +
binary decision diagrams (bdd)

wala

java, conventional approach

bddbddb

datalog + java + bdd



not a single purely declarative specification

coupling of specification and algorithm

our pointer analysis framework

- datalog-based pointer analysis framework for java

DOOP

our pointer analysis framework

- datalog-based pointer analysis framework for java
- declarative: what, not how

DOOP

our pointer analysis framework

- datalog-based pointer analysis framework for java
- declarative: what, not how

  easier to express sophisticated analyses
  correctness more clear
  clear variation points
  eases exploration of approximations
  enables aggressive optimization

DOOP

our pointer analysis framework

- datalog-based pointer analysis framework for java
- declarative: what, not how
  - easier to express sophisticated analyses
  - correctness more clear
  - clear variation points
  - eases exploration of approximations
  - enables aggressive optimization
- sophisticated
  - 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

DOOP

our pointer analysis framework

- datalog-based pointer analysis framework for java
- declarative: what, not how
  - easier to express sophisticated analyses
  - correctness more clear
  - clear variation points
  - eases exploration of approximations
  - enables aggressive optimization
- sophisticated
  - 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

DOOP

our pointer analysis framework

- datalog-based pointer analysis framework for java
- declarative: what, not how
  - easier to express sophisticated analyses
  - correctness more clear
  - clear variation points
  - eases exploration of approximations
  - enables aggressive optimization
- sophisticated
  - 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
- enables precision and performance comparison

DOOP

expressed pointer analysis in full sophistication in datalog

- core specification: ~250 lines of logic
- full specification: ~2500 lines of logic

key contributions

expressed pointer analysis in full sophistication in datalog

- core specification: ~250 lines of logic
- full specification: ~2500 lines of logic

synthesized efficient algorithms from specification

- order of magnitude performance improvement
- support all analyses considered interesting

key contributions

expressed pointer analysis in full sophistication in datalog

- core specification: ~250 lines of logic
- full specification: ~2500 lines of logic

synthesized efficient algorithms from specification

- order of magnitude performance improvement
- support all analyses considered interesting

approach: heuristics for searching algorithm space

- targeted at recursive problem domains

key contributions

expressed pointer analysis in full sophistication in datalog

- core specification: ~250 lines of logic
- full specification: ~2500 lines of logic

synthesized efficient algorithms from specification

- order of magnitude performance improvement
- support all analyses considered interesting

approach: heuristics for searching algorithm space

- targeted at recursive problem domains

demonstrated scalability with explicit representation

our contributions are surprising

expressed pointer analysis in full sophistication in datalog

our contributions are surprising

expressed pointer analysis in full sophistication 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."*

our contributions are surprising

expressed pointer analysis in full sophistication 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

our contributions are surprising

expressed pointer analysis in full sophistication 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"*

our contributions are surprising

expressed pointer analysis in full sophistication 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"*

our contributions are surprising

expressed pointer analysis in full sophistication 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."*

from algorithms to specification

pointer analysis and datalog background

var points-to

program analysis: a domain of mutual recursion 14

```
x = y
```

var points-to

program analysis: a domain of mutual recursion 14

```
x = y
```

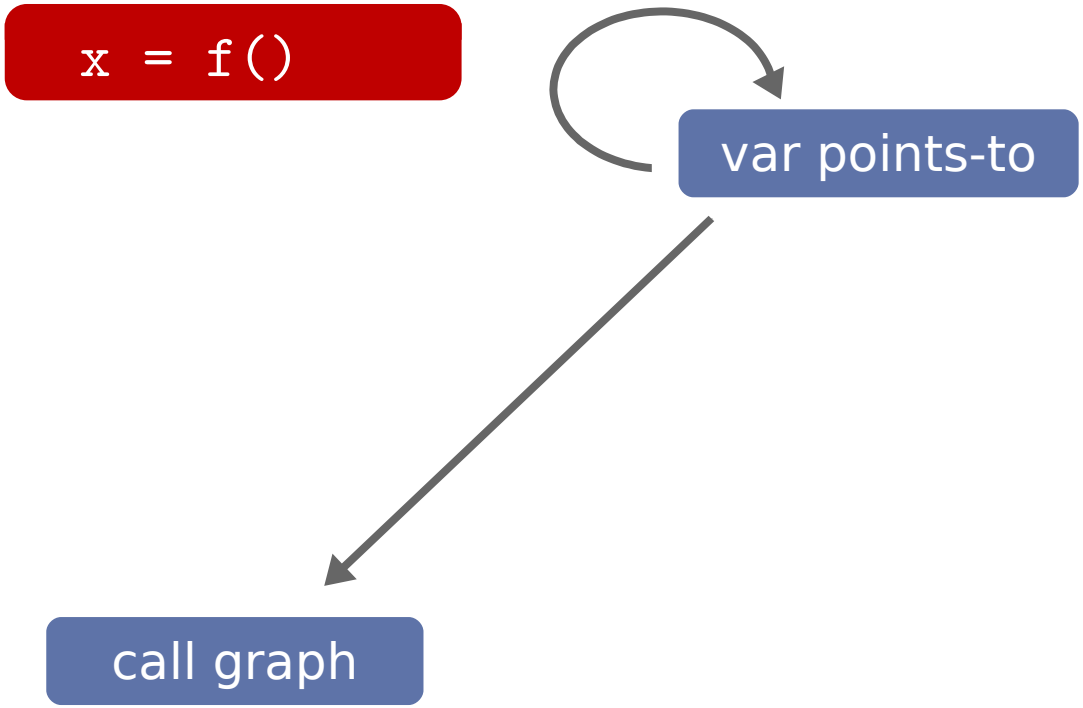
program analysis: a domain of mutual recursion 14

```
x = f()
```

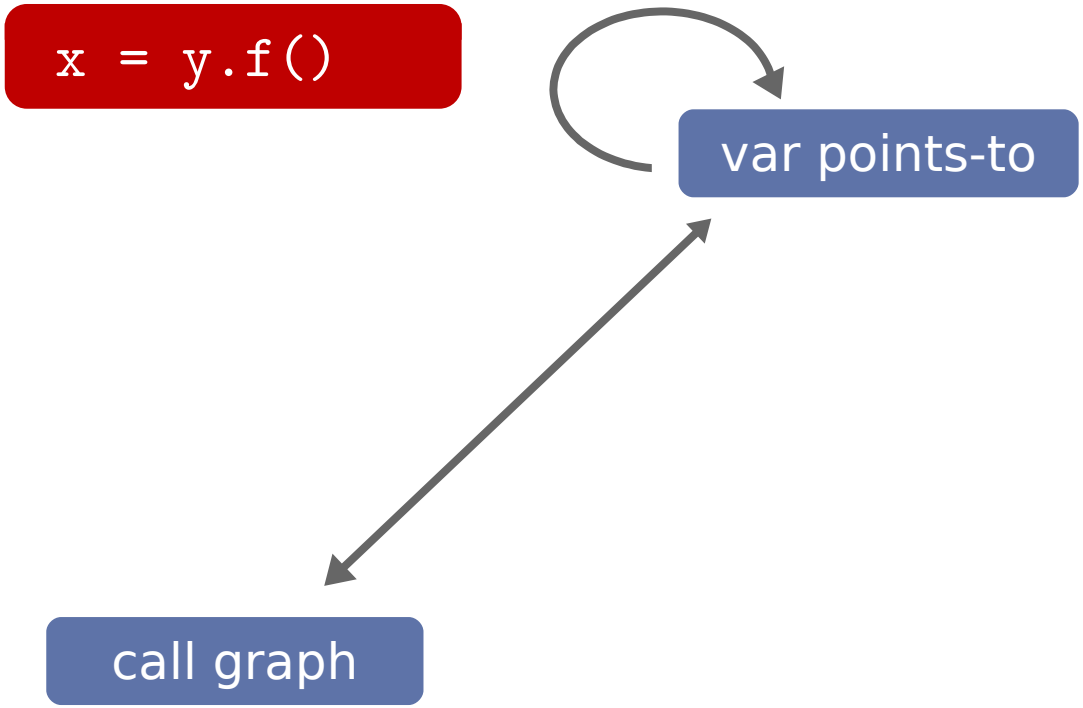
var points-to

call graph

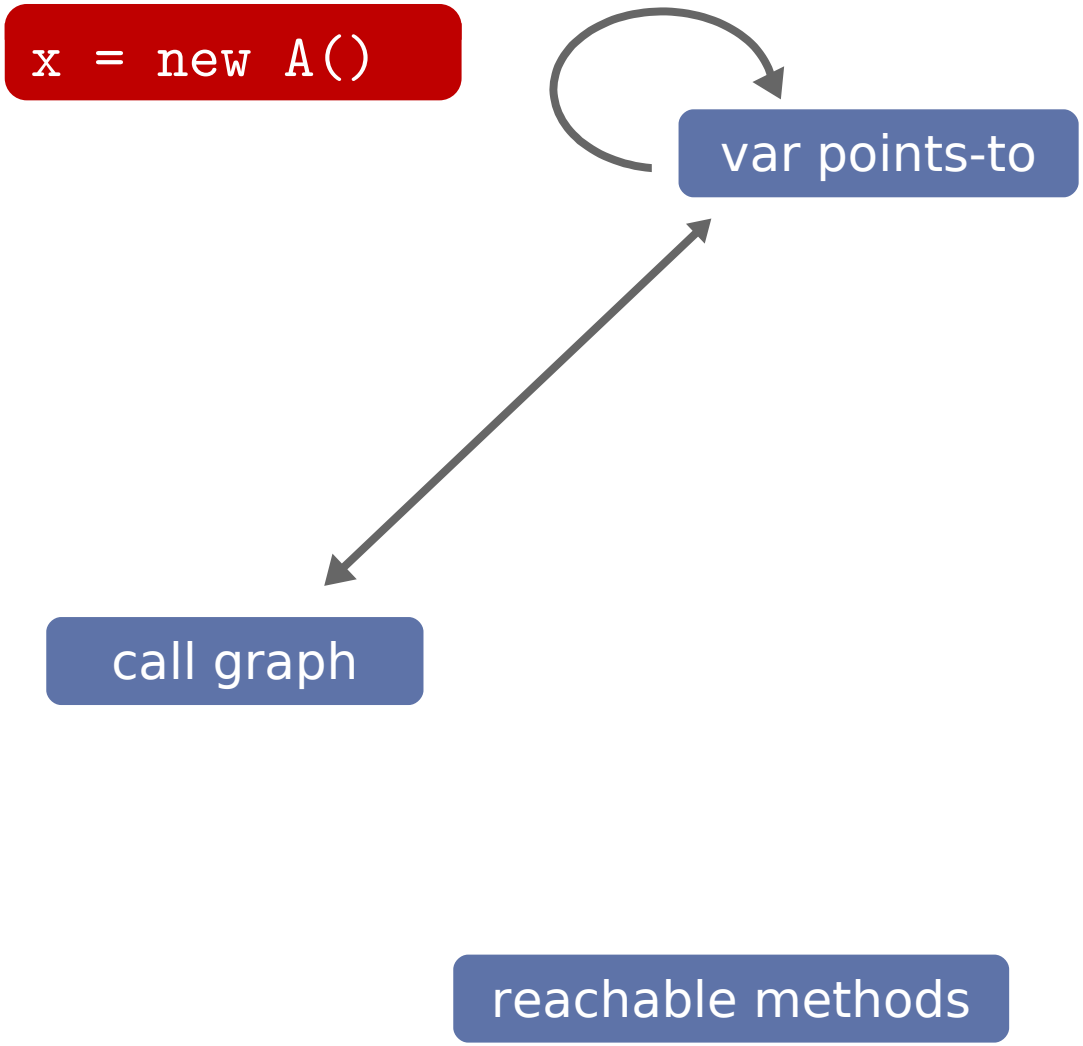
program analysis: a domain of mutual recursion 14



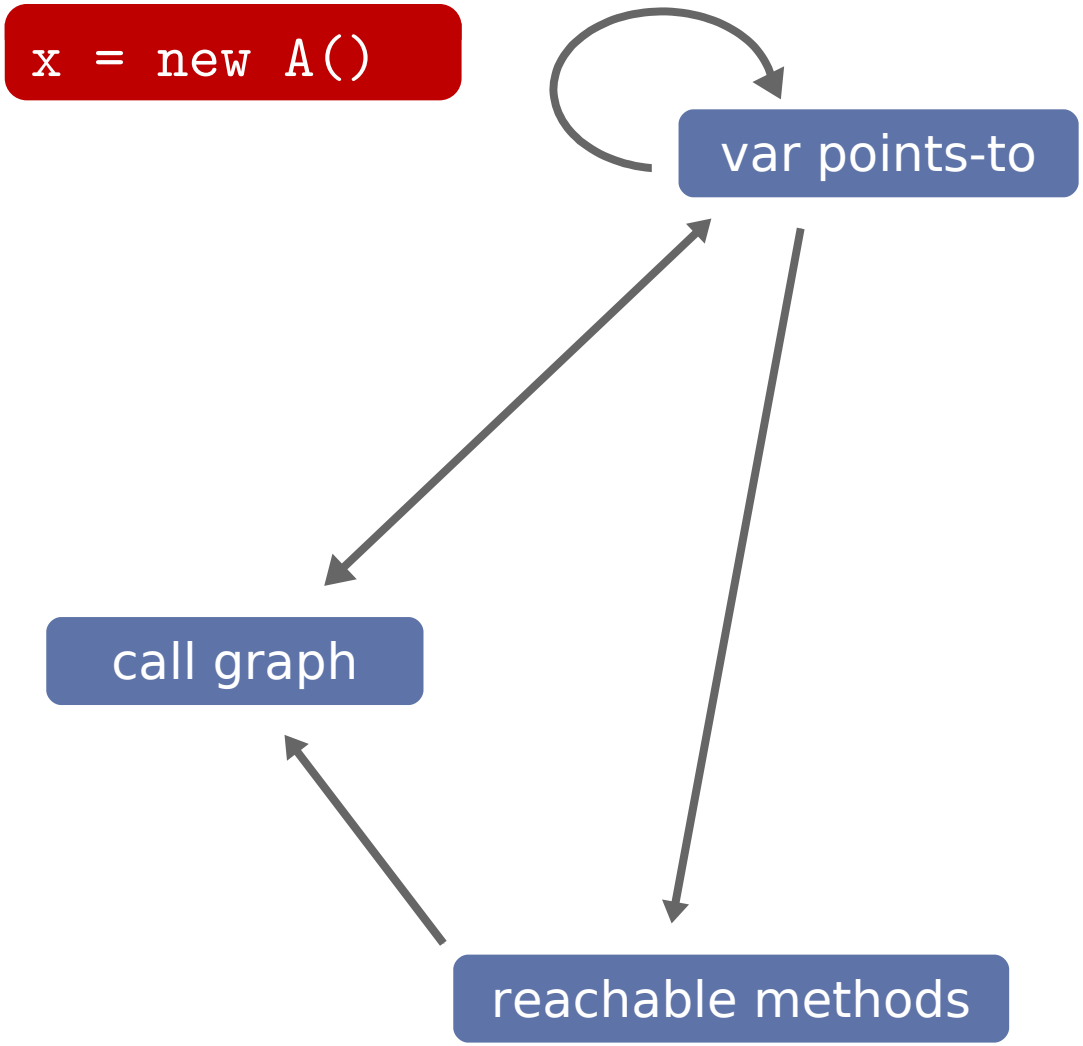
program analysis: a domain of mutual recursion 14



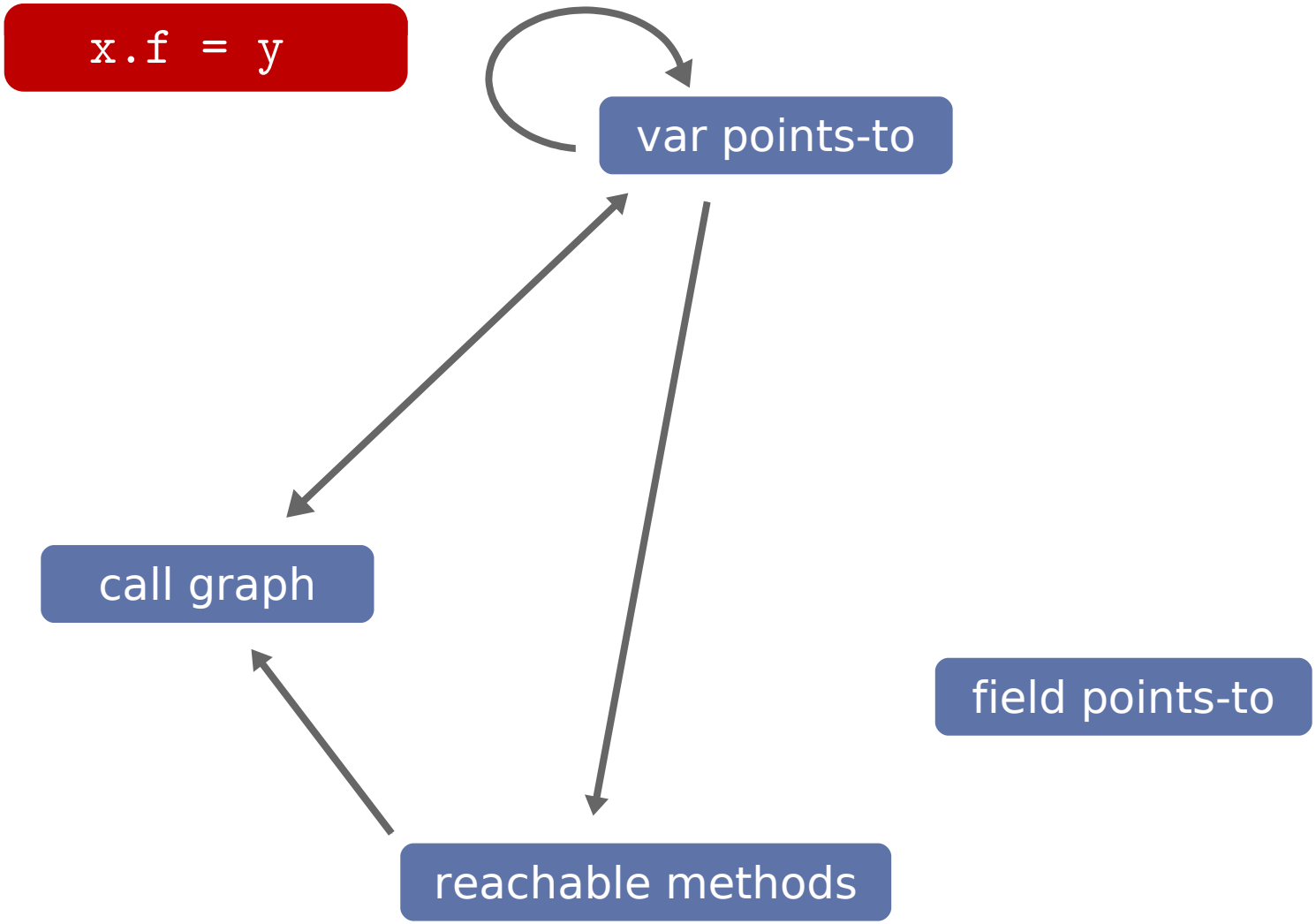
program analysis: a domain of mutual recursion 14



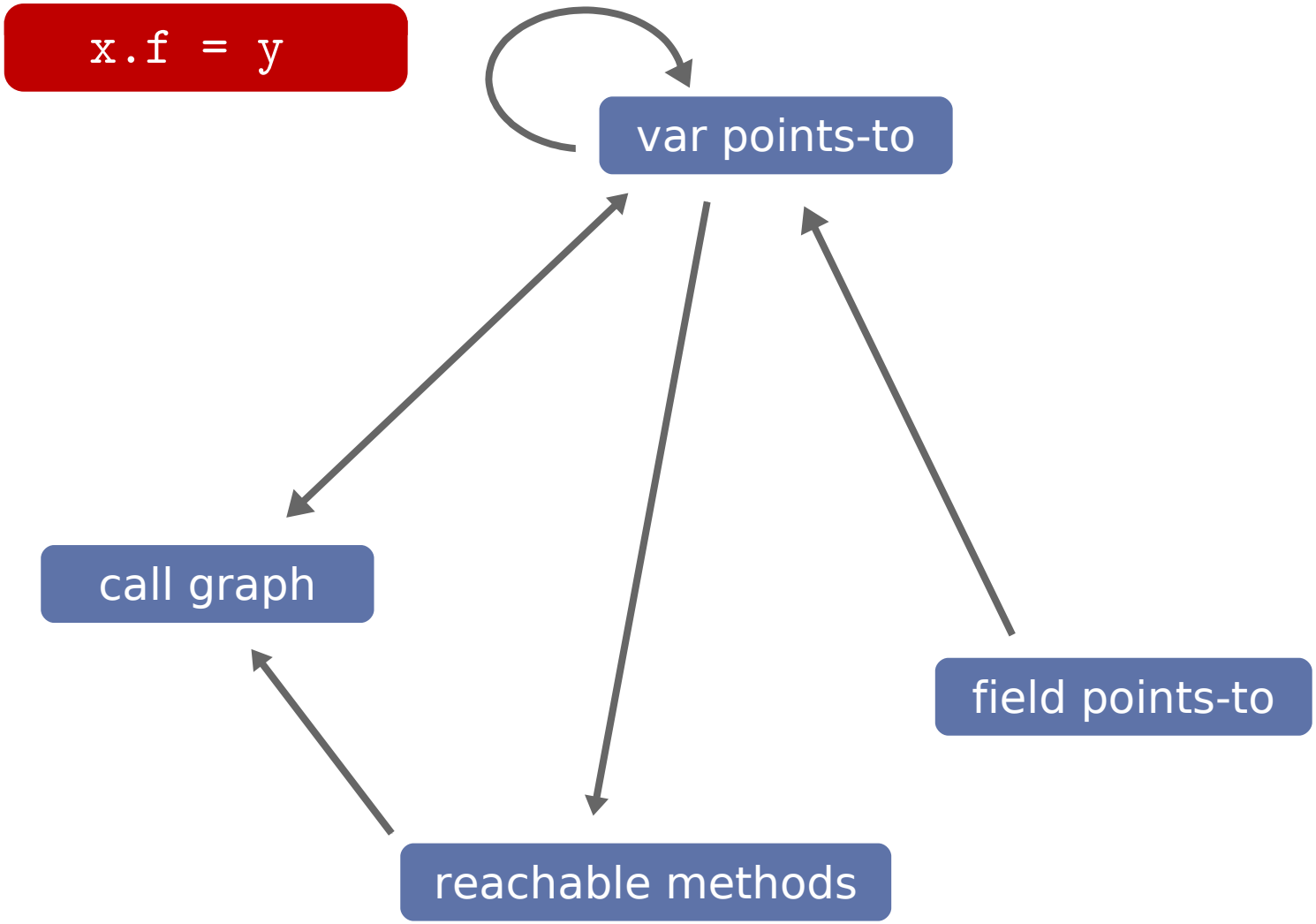
program analysis: a domain of mutual recursion 14



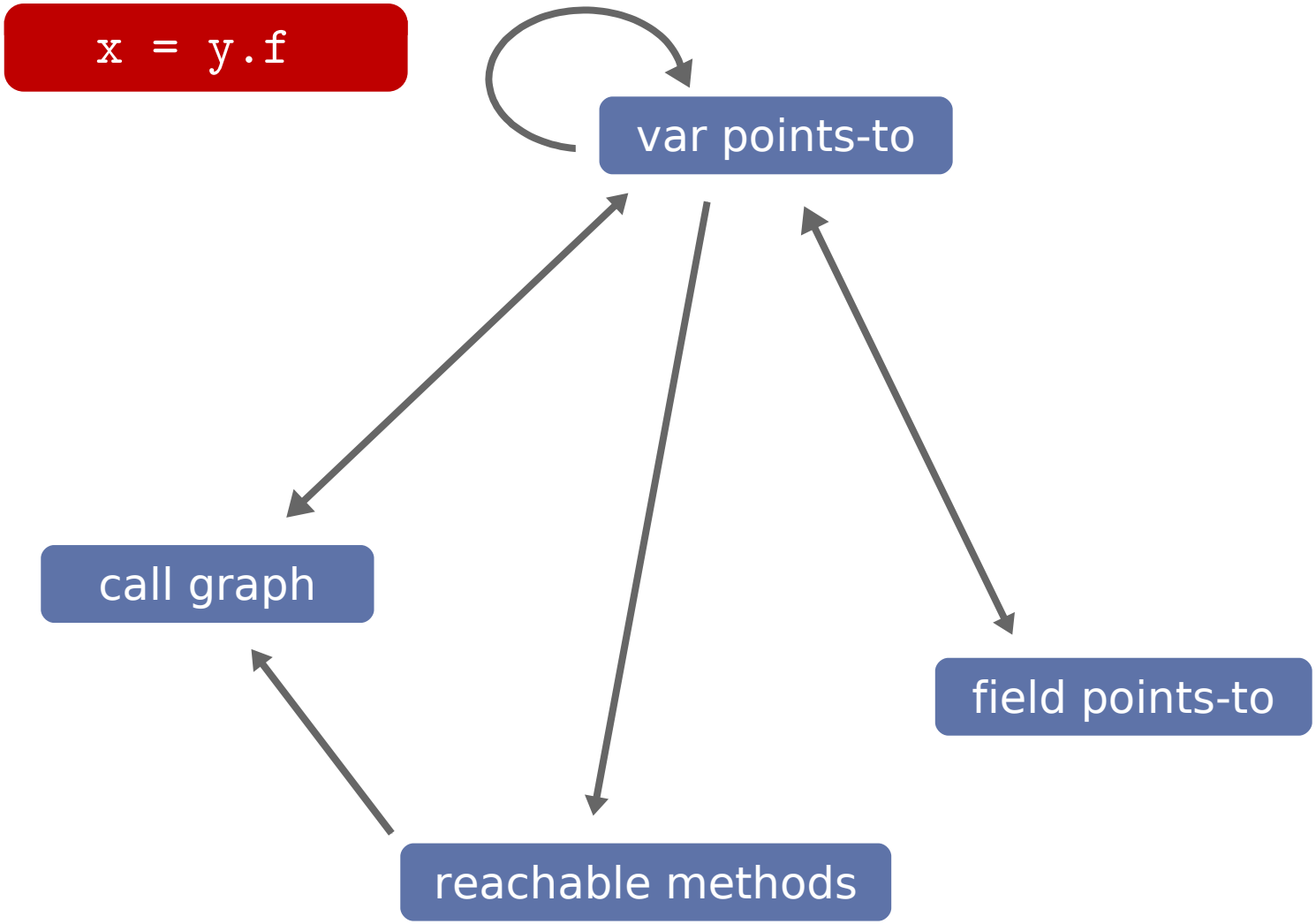
program analysis: a domain of mutual recursion 14



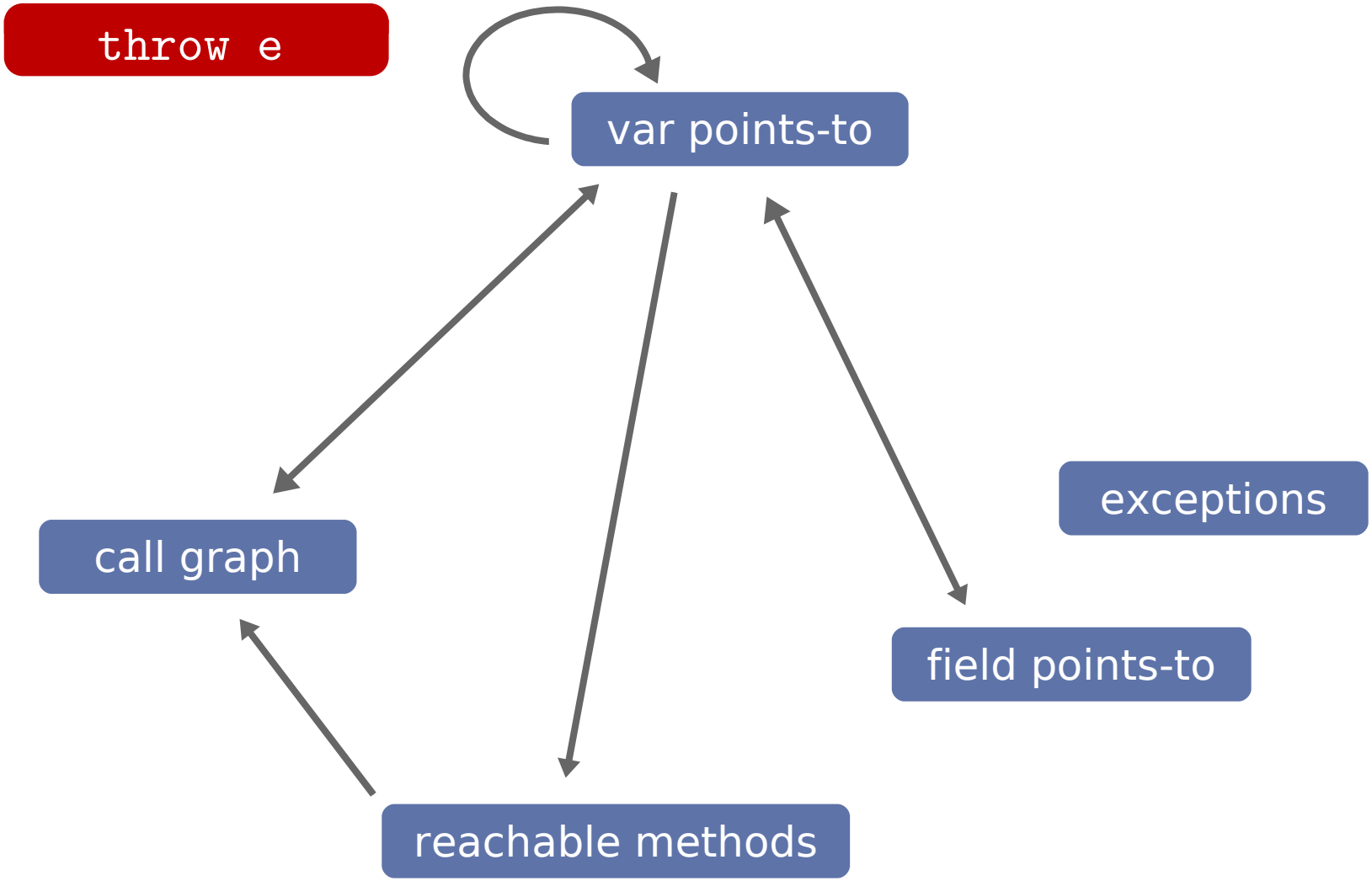
program analysis: a domain of mutual recursion 14



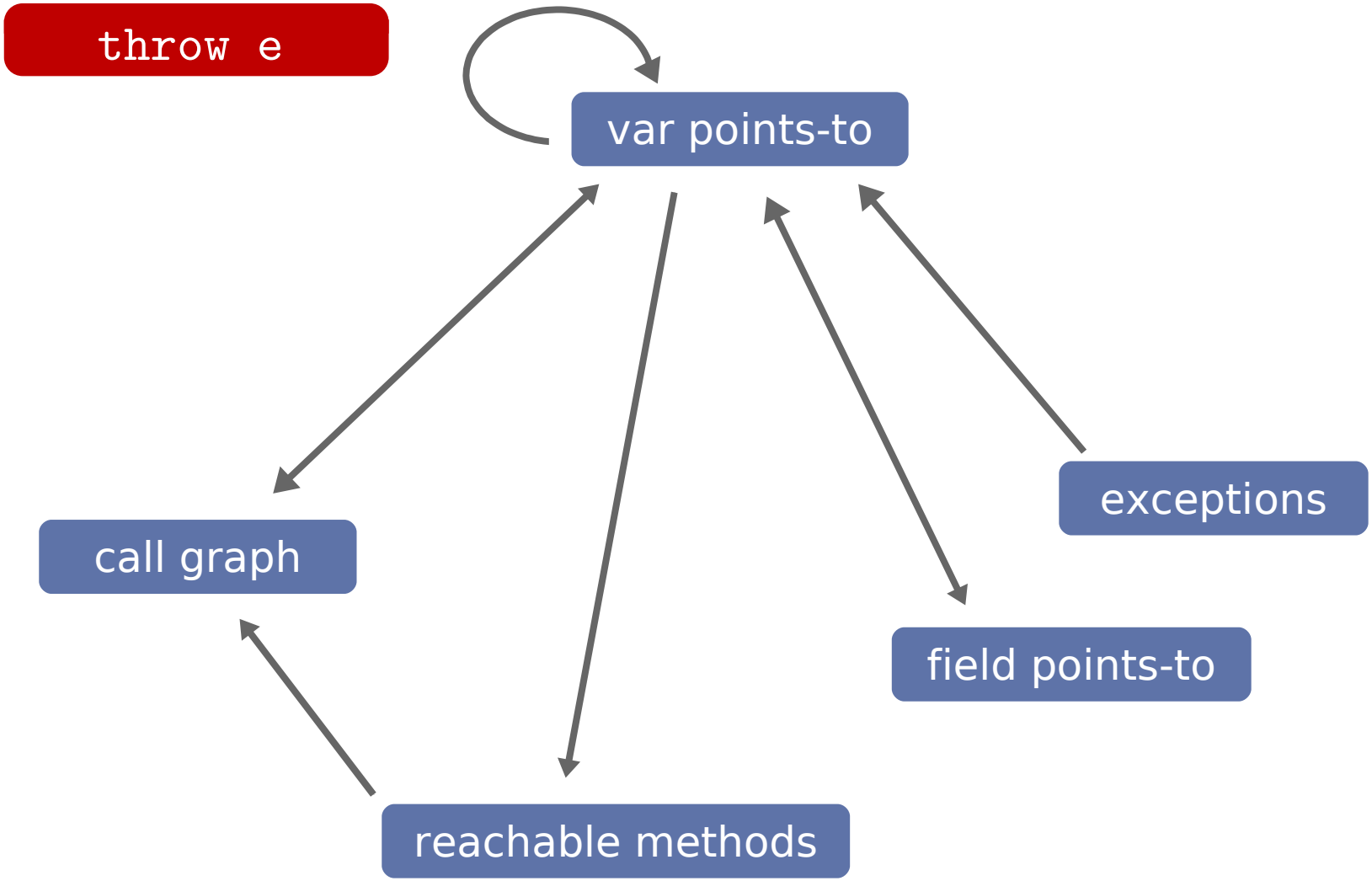
program analysis: a domain of mutual recursion 14



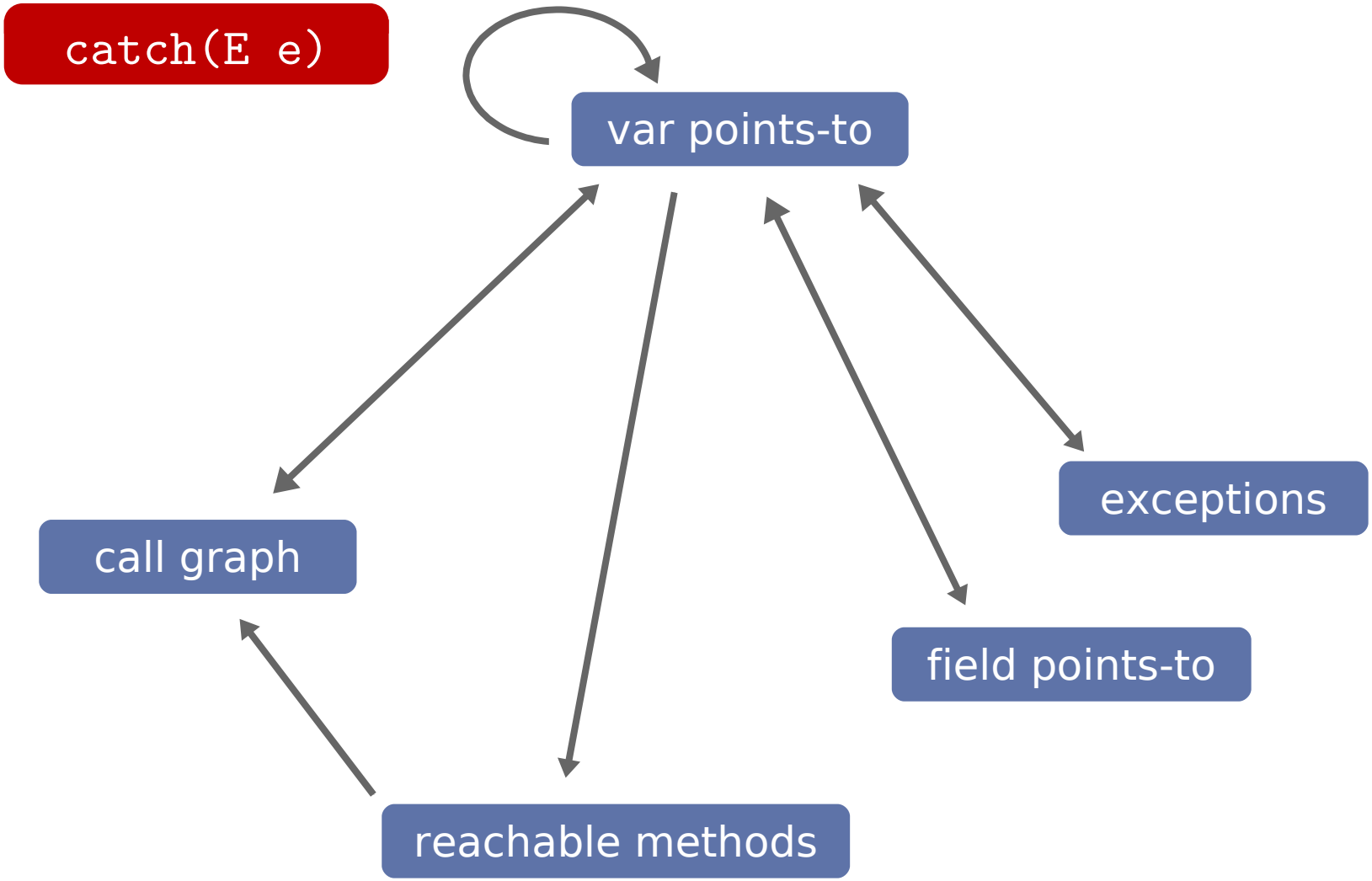
program analysis: a domain of mutual recursion 14



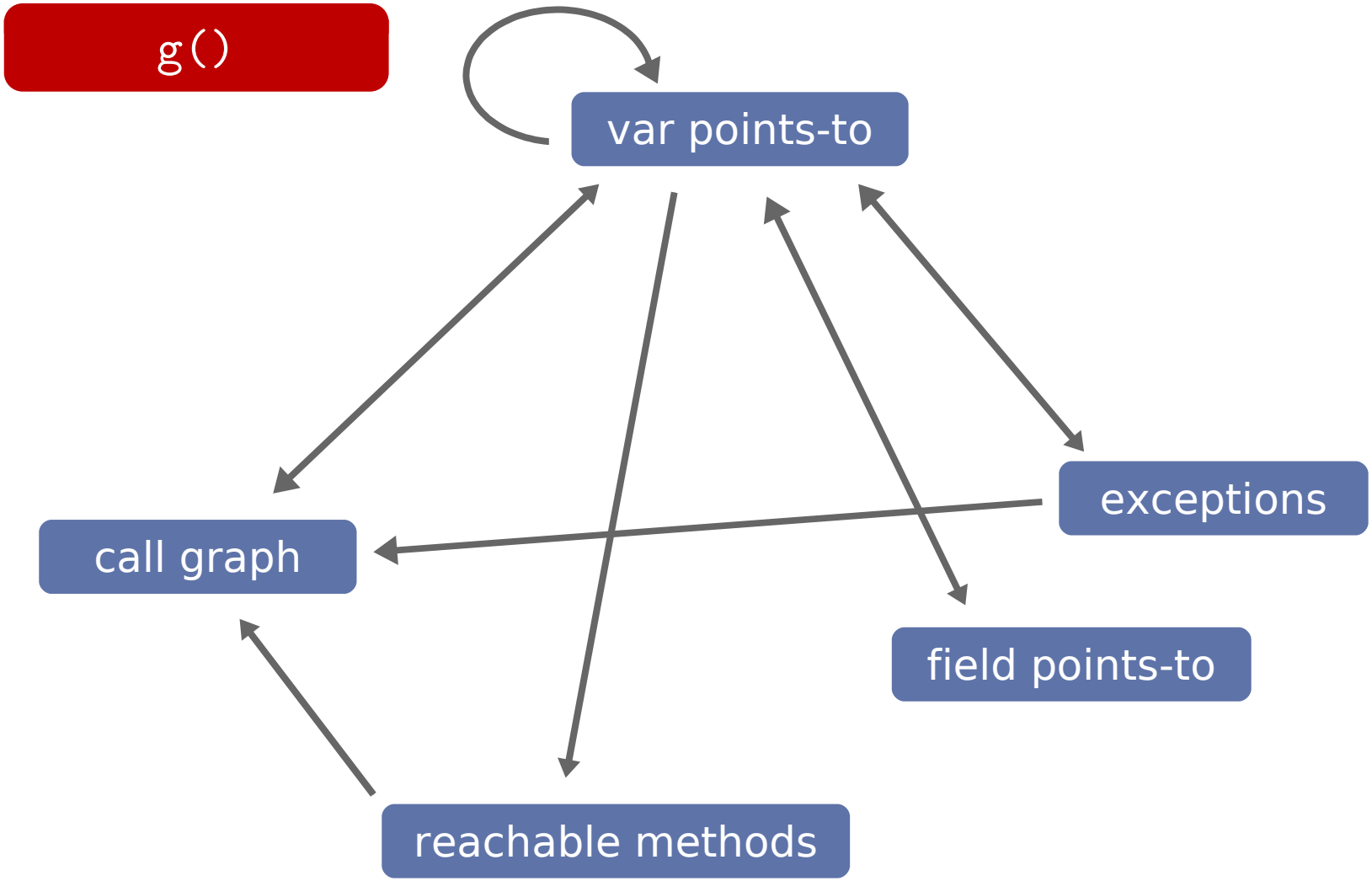
program analysis: a domain of mutual recursion 14



program analysis: a domain of mutual recursion 14



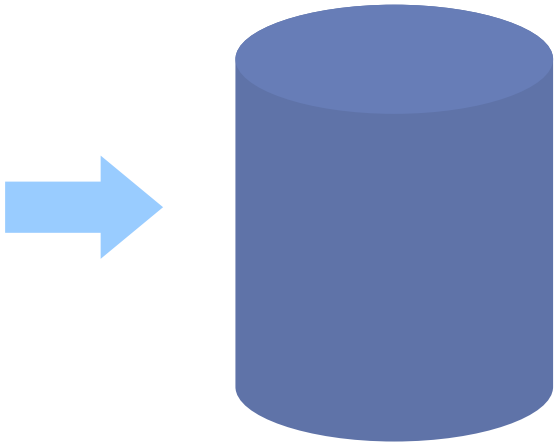
program analysis: a domain of mutual recursion 14



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;
```

AssignObjectAllocation

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;
```

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

Assign

b	a
a	b
b	c

VarPointsTo

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

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

datalog: declarative mutual recursion

source

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

AssignObjectAllocation

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

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  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
  - enables more aggressive optimization

writing datalog is less programming, more specification

from algorithms to specification

declarative pointer analysis specification

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

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

example 1: introducing fields

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

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

`?base . ?field = ?from`

example 1: introducing fields


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

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

FieldPointsTo(?baseobj, ?field, ?obj) <-
```

?base . ?field = ?from

example 1: introducing fields


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

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

FieldPointsTo(?baseobj, ?field, ?obj) <-
```

field `?field`
of object `?baseobj`
may point to object `?obj`

`?base . ?field = ?from`

example 1: introducing fields


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

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

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

field `?field`
of object `?baseobj`
may point to object `?obj`

`?base . ?field = ?from`

example 1: introducing fields


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

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

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

field ?field
of object ?baseobj
may point to object ?obj

?base . ?field = ?from

?baseobj ?obj

example 1: introducing fields


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

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

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

```
?to = ?base . ?field
```

example 1: introducing fields


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

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

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

VarPointsTo(?to, ?obj) <-
```

?to = ?base . ?field

example 1: introducing fields


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

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

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

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

?to = ?base . ?field

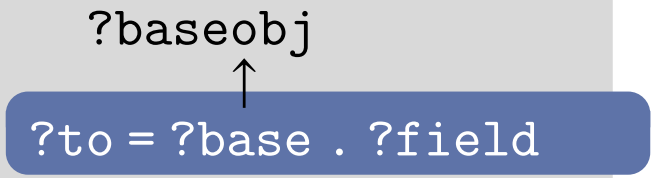
example 1: introducing fields

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

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

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

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



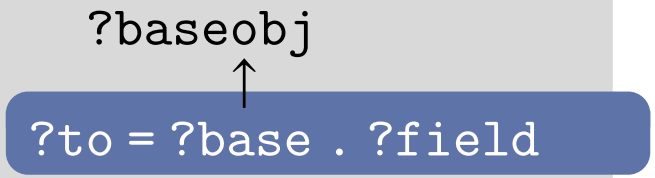
example 1: introducing fields


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

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

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

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



example 1: introducing fields


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

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

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

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

modularity: introducing new mutually recursive dependencies does not change existing rules

method invocations: propagated exceptions

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

example 2: precise exception analysis

method invocations: propagated exceptions

- ```
  ThrowPointsTo(?caller, ?obj) <-
  ```

```
void f() {
  g();
}
```

method invocations: caught exceptions

- ```
  VarPointsTo(?param, ?obj) <-
  ```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

example 2: precise exception analysis

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

example 2: precise exception analysis

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

example 2: precise exception analysis

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  ExceptionHandler[?objtype, ?invocation] = ?handler,
```

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

example 2: precise exception analysis

method invocations: propagated exceptions

```
void f() {
  g();
}
```

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Method[?invocation] = ?caller.
```

method invocations: caught exceptions

```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

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

example 2: precise exception analysis

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

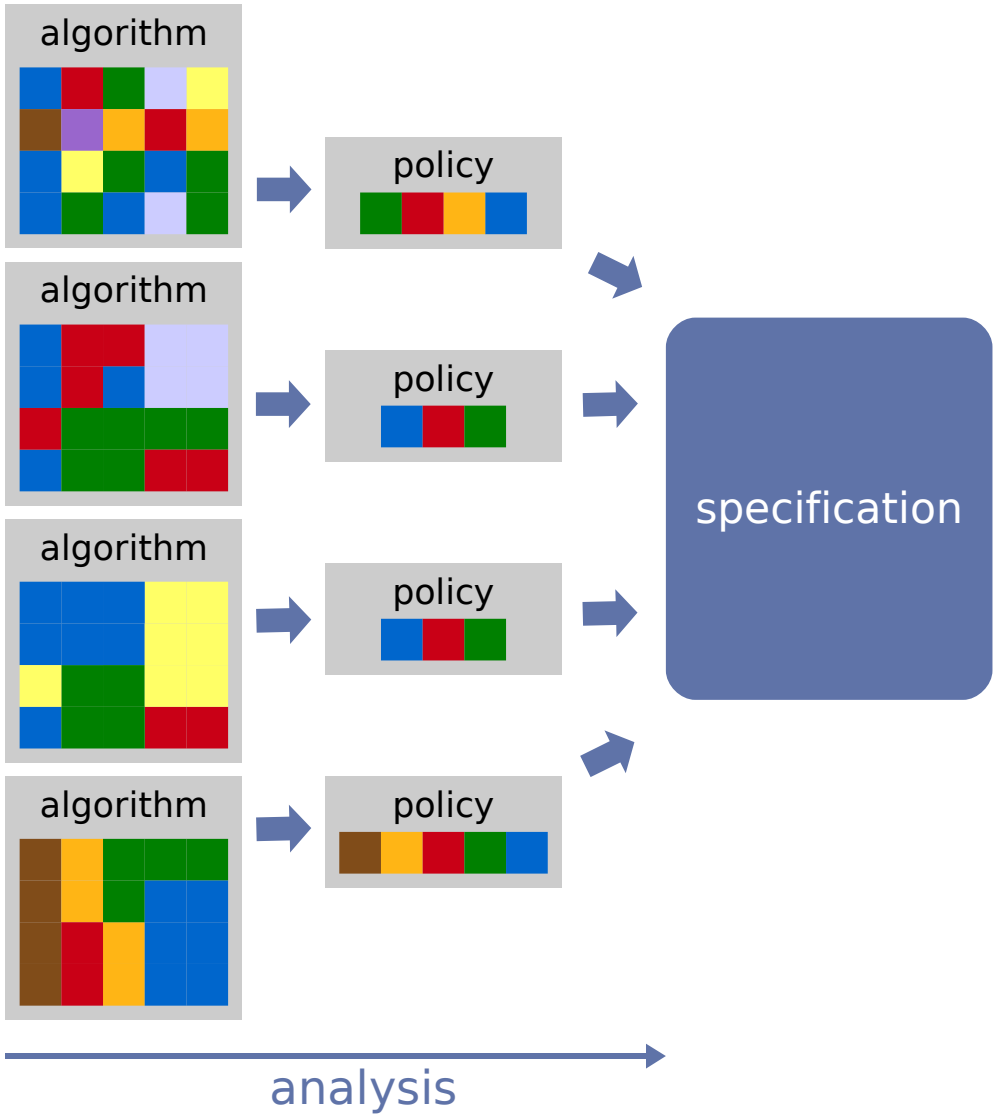
method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  [illegible]
  ype, ?invocation] = ?handler,
  lParam[?handler] = ?param.
```

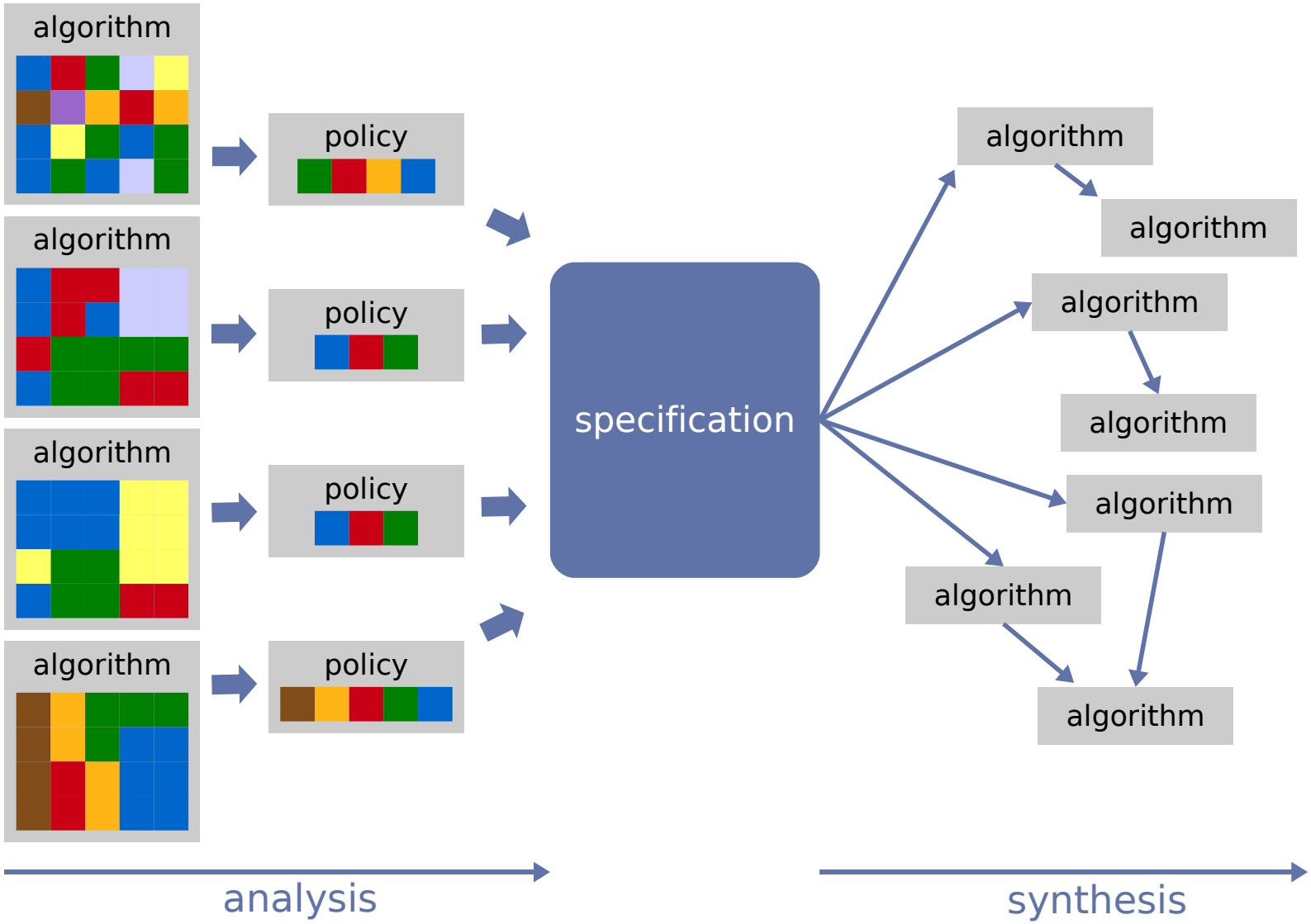
```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

again, new mutually recursive dependencies do not change existing rules

from algorithms to specification and back



from algorithms to specification and back

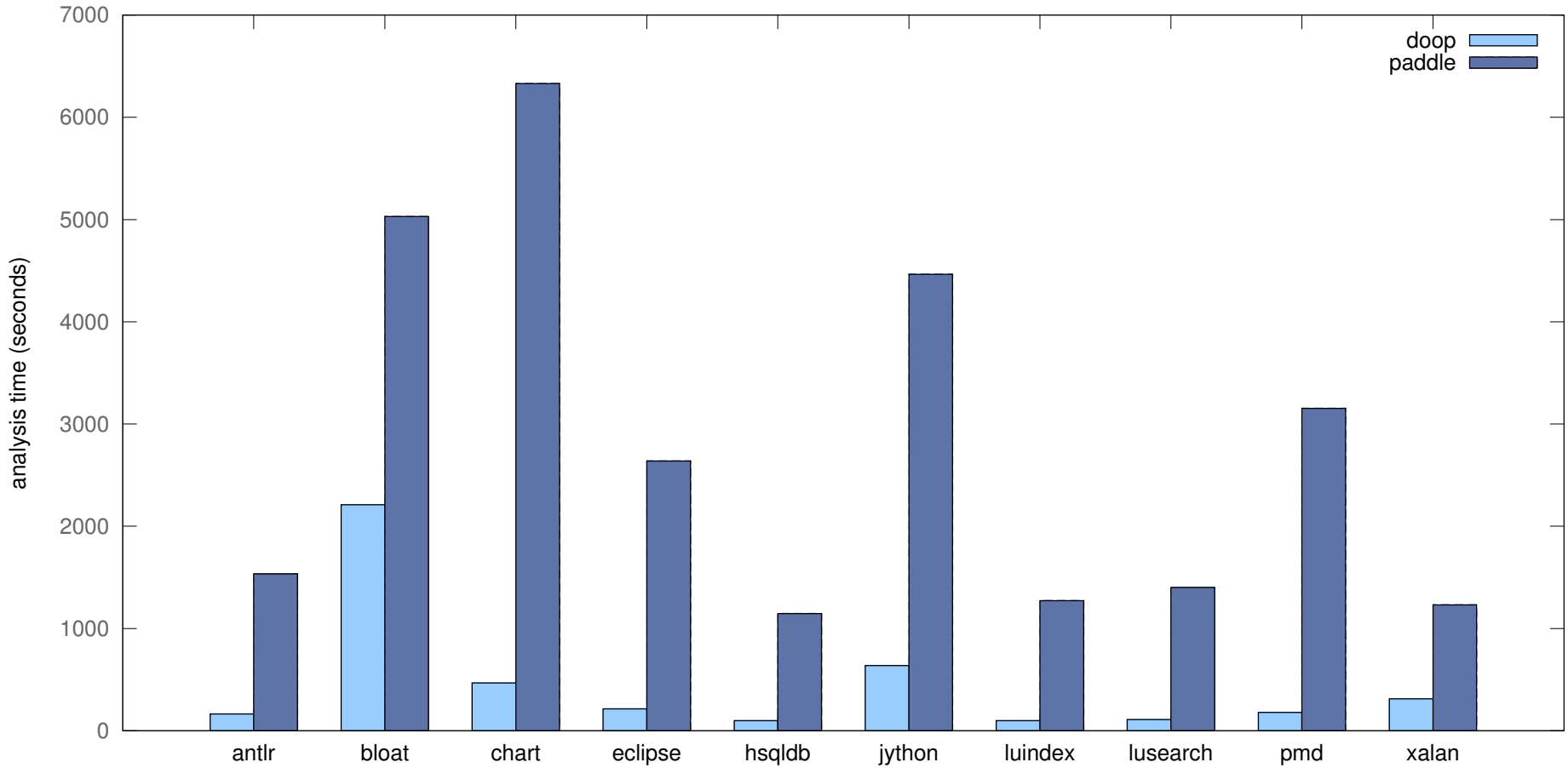


from specification to algorithms

from specification to algorithms

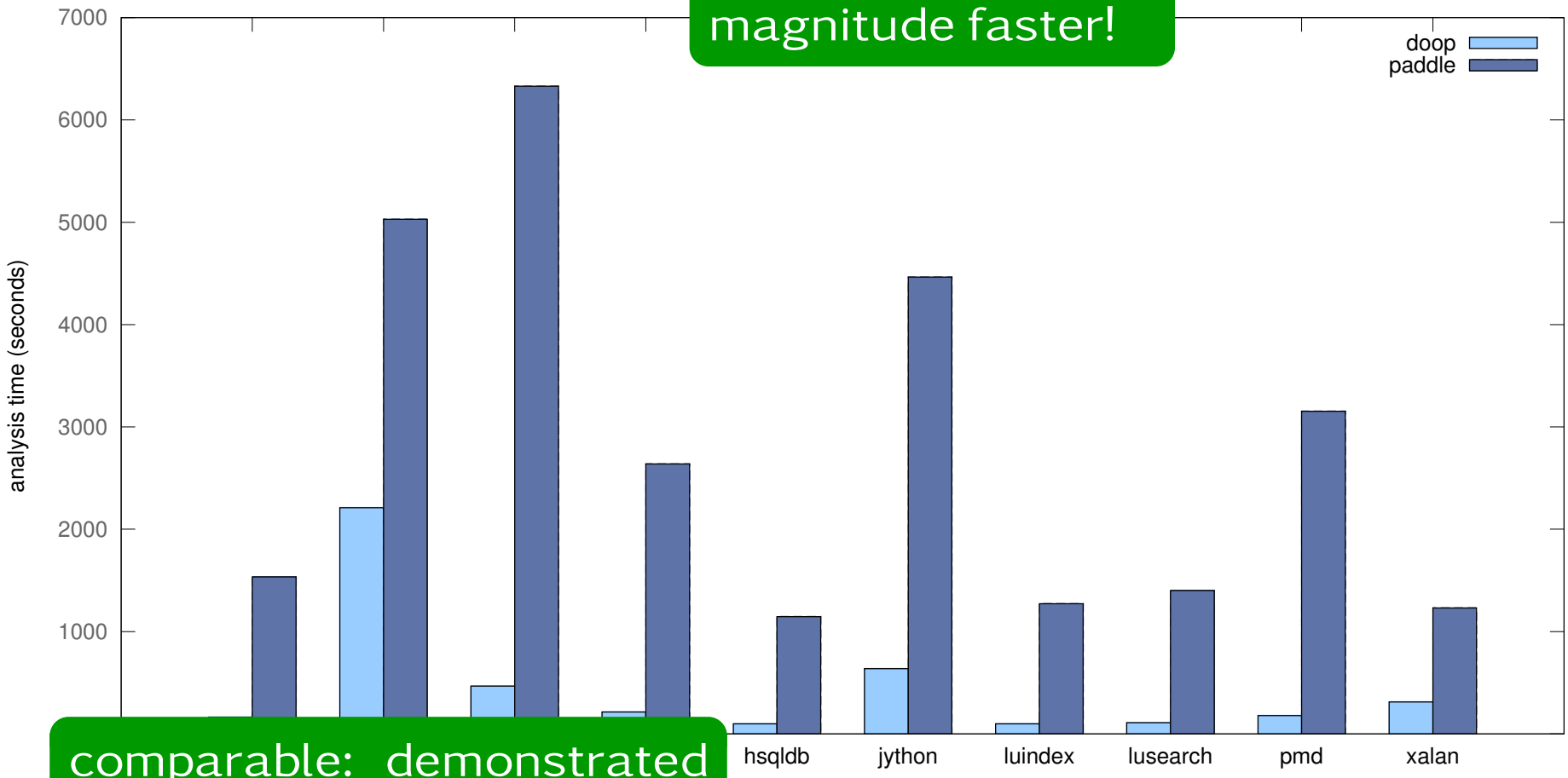
benchmark

benchmark: 1-call-site-sensitive+heap



benchmark: 1-call-site-sensitive+heap

full order of magnitude faster!



comparable: demonstrated equivalence of results!

where is the magic?

strictly declarative

- no coupling of specification and algorithm

efficient architecture for datalog engine

- but, not a magic tool

$\Rightarrow$ novel heuristics for searching the algorithm space

- targeting recursive logic

from specification to algorithms

datalog evaluation background

datalog

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

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

naive evaluation (relational algebra)

VarPointsTo := AssignObjectAllocation

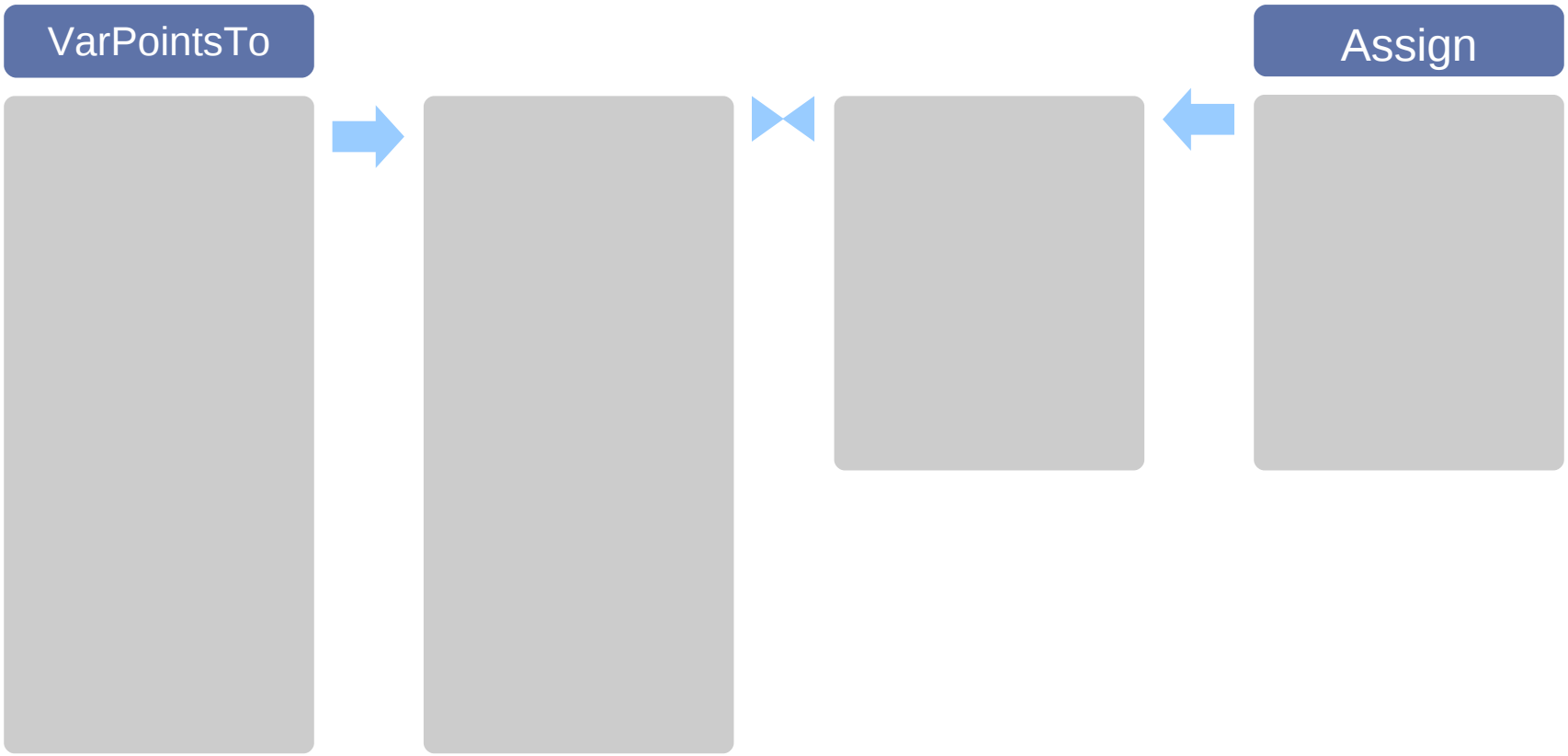
repeat

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

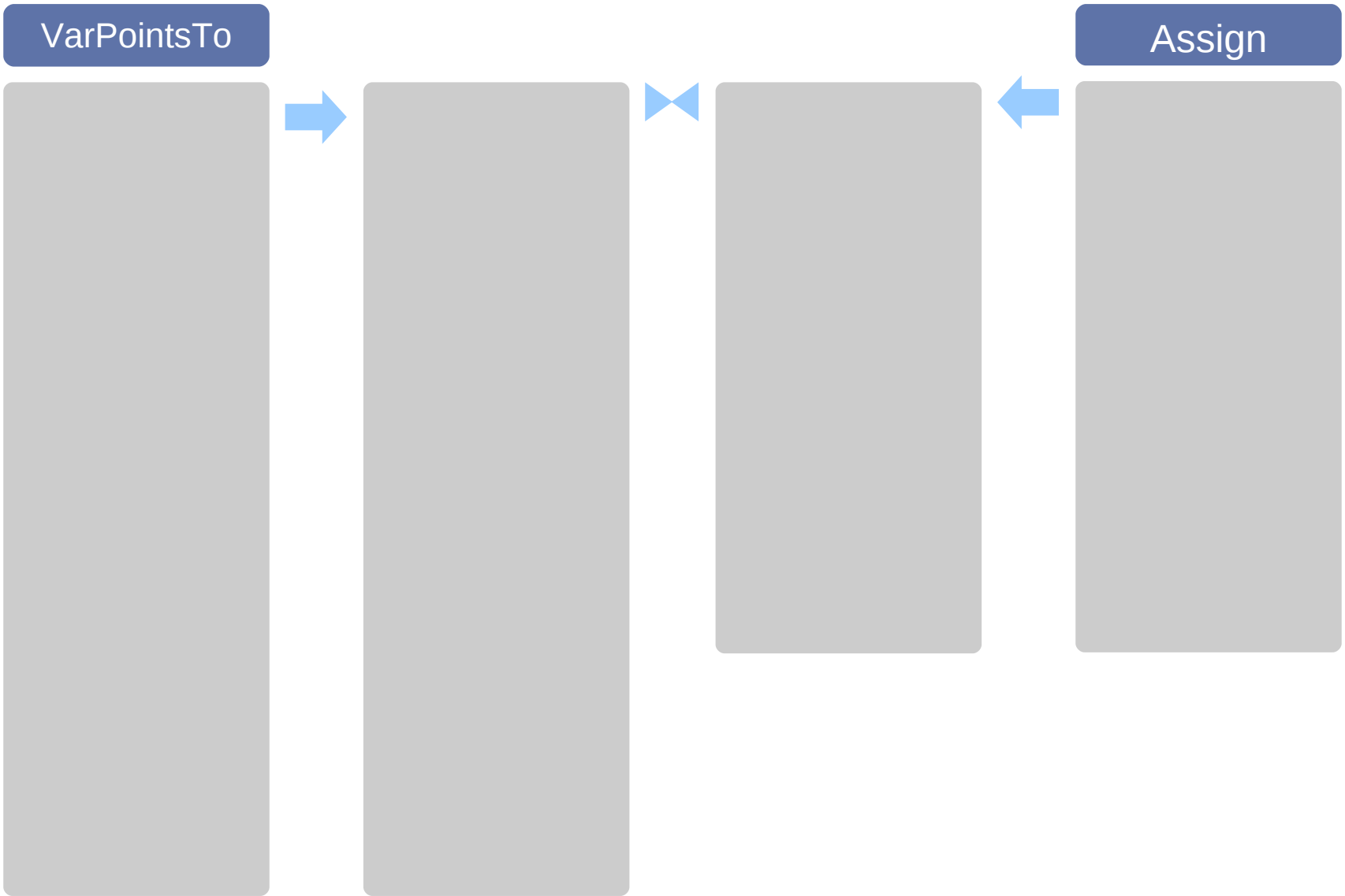
  VarPointsTo := VarPointsTo $\cup$ tmp

until fixpoint

datalog: naive evaluation

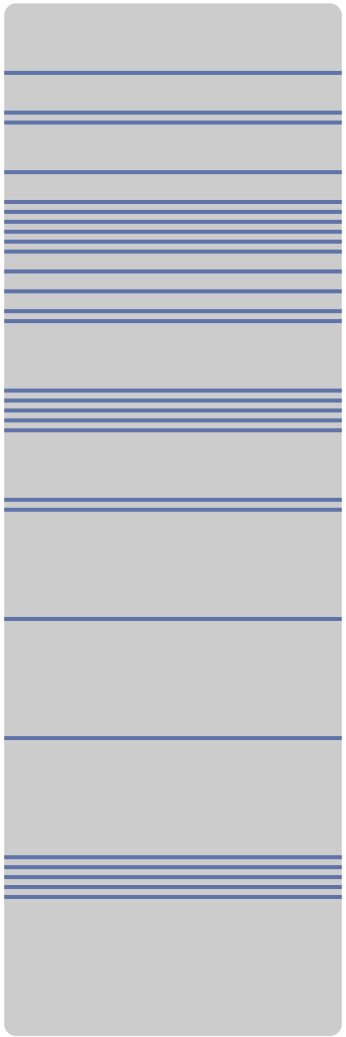


datalog: naive evaluation

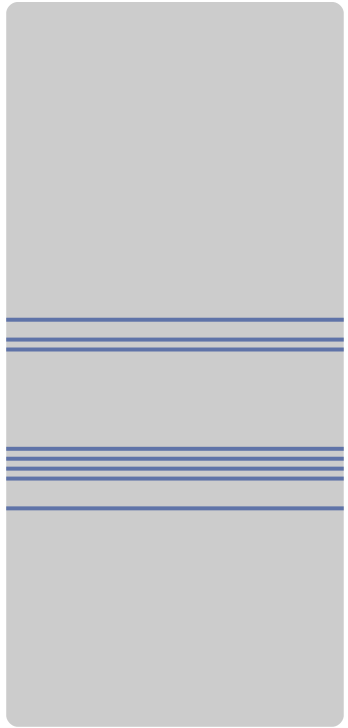


datalog: semi-naive evaluation

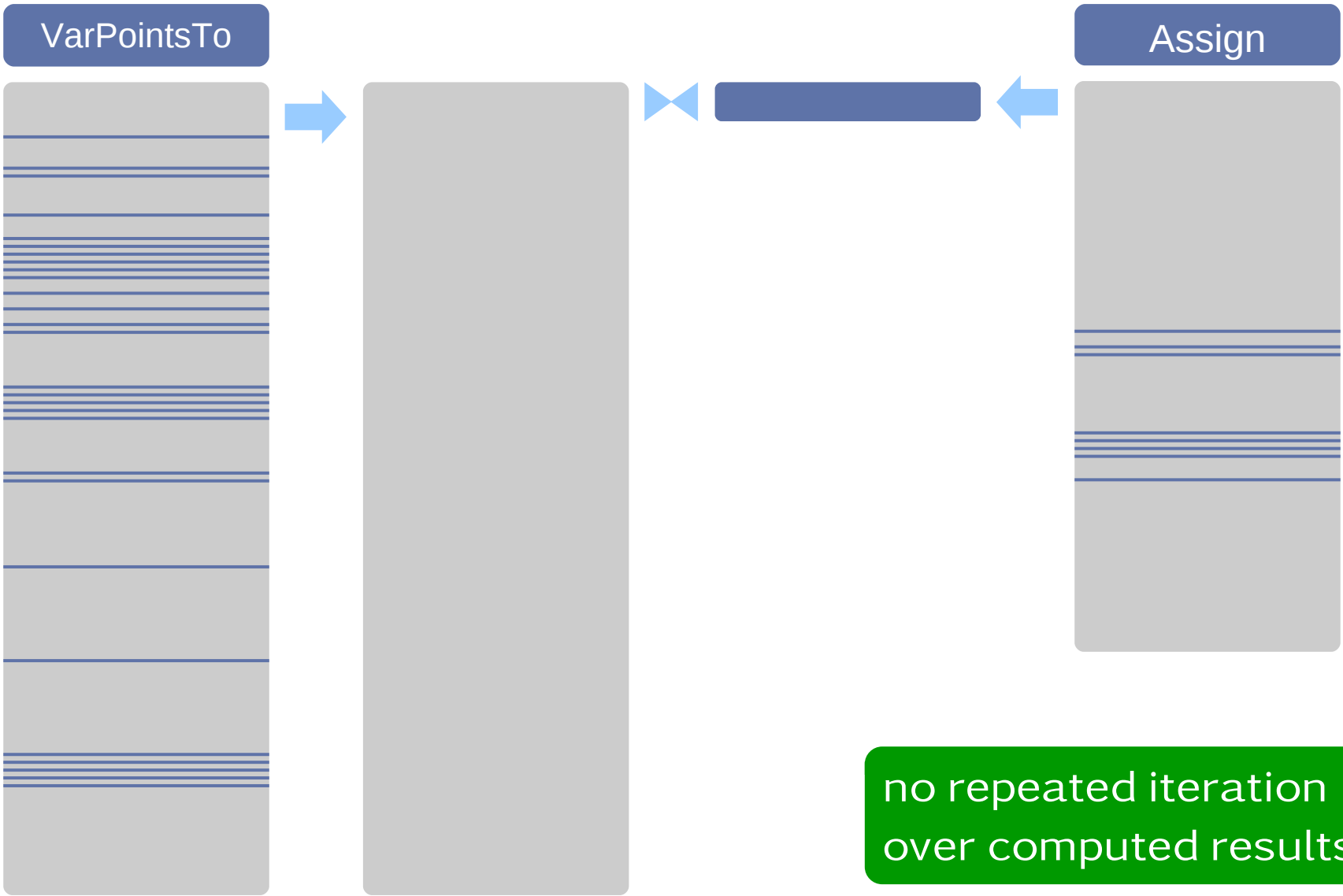
VarPointsTo



Assign

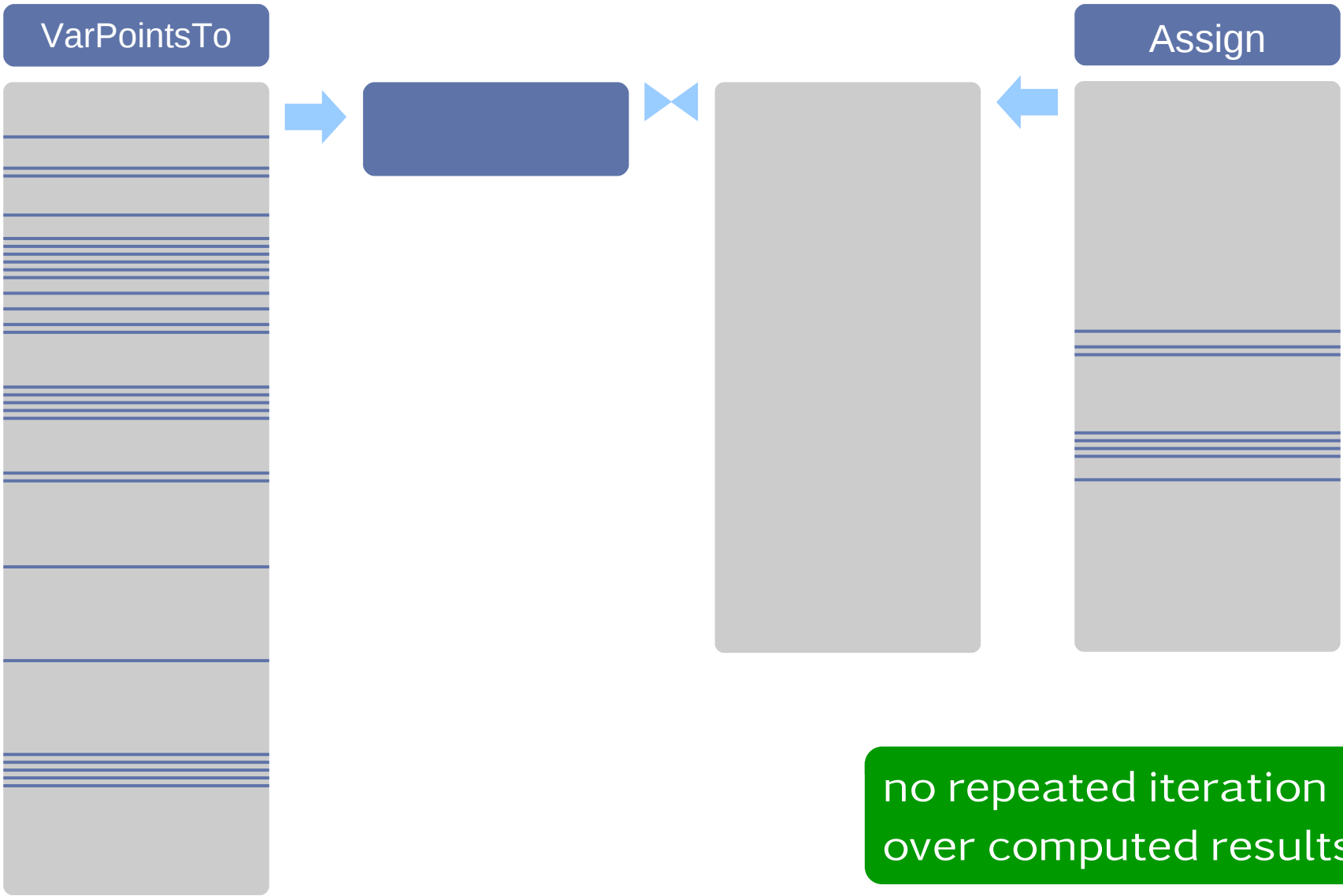


datalog: semi-naive evaluation



no repeated iteration
over computed results

datalog: semi-naive evaluation



no repeated iteration
over computed results

from specification to algorithms

datalog engine architecture

datalog engine architecture

efficient low-level representation

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

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

relations are multidimensional indexes (b-trees)

'index-organized tables'

efficient representation of sparse relations

alternative data structures for (partially) dense relations

indexes exposed in the language

argument order determines index

enables us to stay in datalog

LOGICBLOX®

relations as multidimensional indexes

VarPointsTo

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

relations as multidimensional indexes

VarPointsTo

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

index organized by
reverse argument order

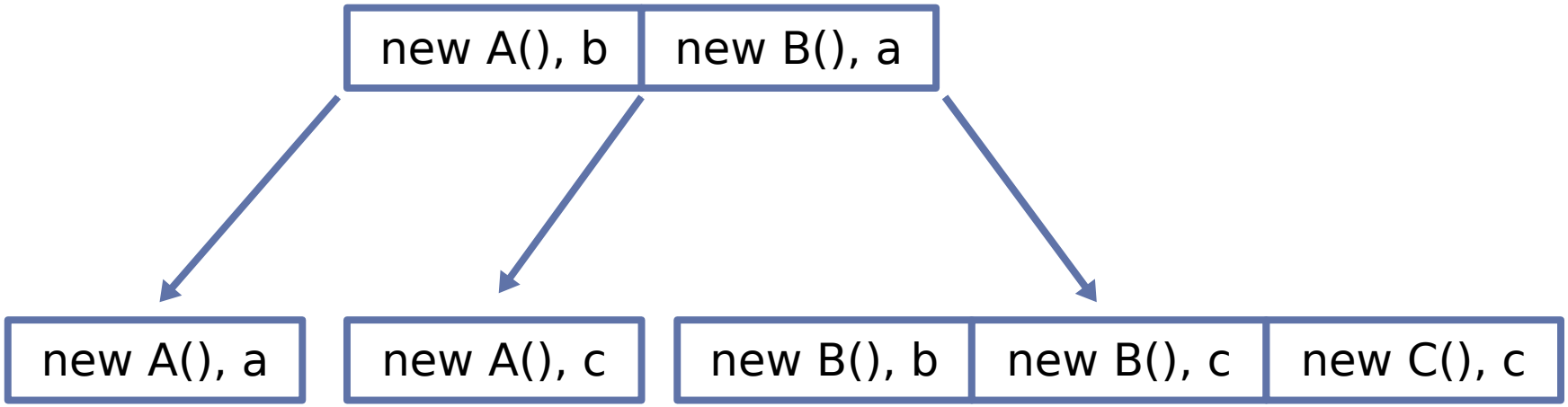
```
VarPointsTo(?var, ?obj)
```

VarPointsTo

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

index organized by
reverse argument order

```
VarPointsTo(?var, ?obj)
```



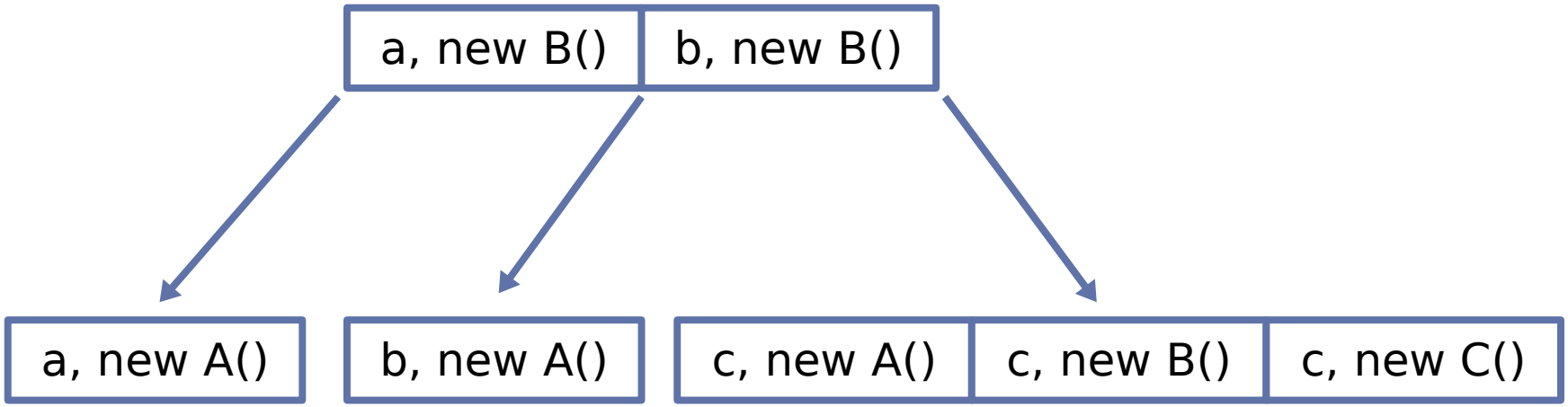
relations as multidimensional indexes

VarPointsTo

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

index organized by
reverse argument order

- `VarPointsTo(?var, ?obj)`
- `VarPointsTo(?obj, ?var)`

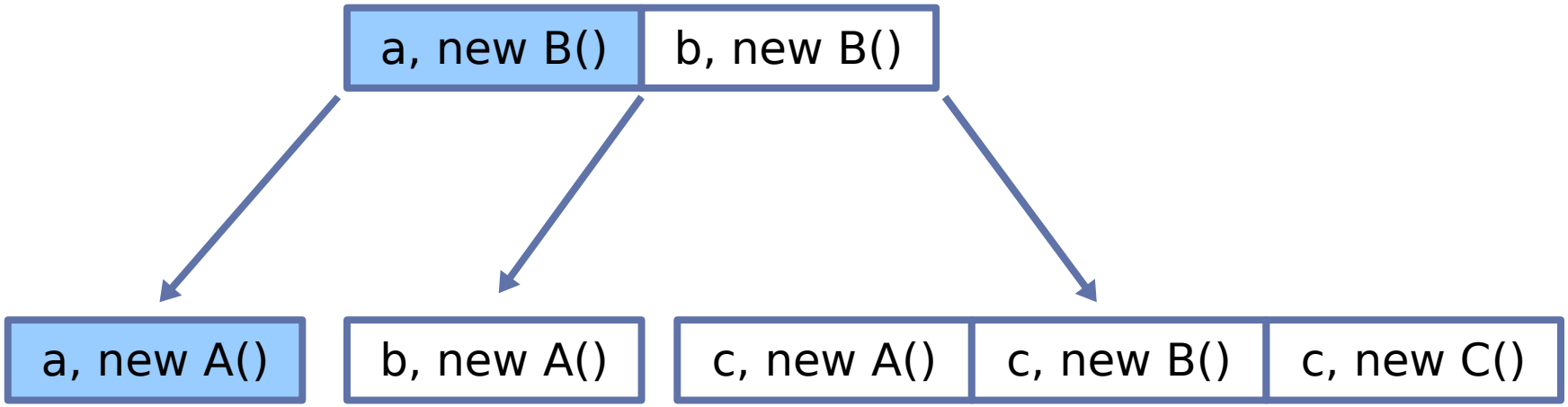


VarPointsTo

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

index organized by
reverse argument order

- `VarPointsTo(?var, ?obj)`
- `VarPointsTo(?obj, ?var)`



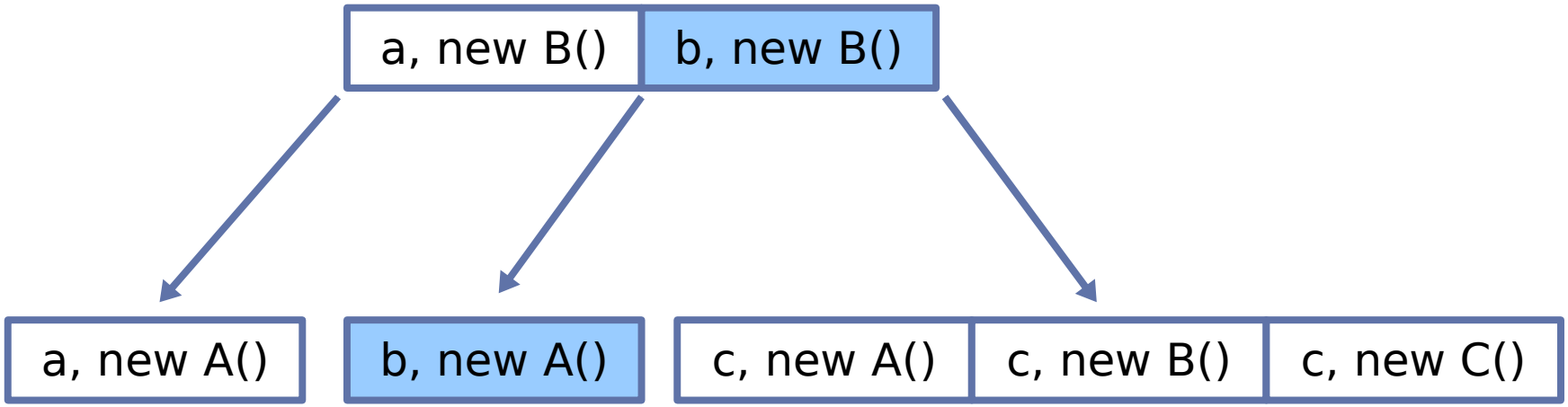
VarPointsTo

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

index organized by
reverse argument order

```
VarPointsTo(?var, ?obj)
```

- `VarPointsTo(?obj, ?var)`



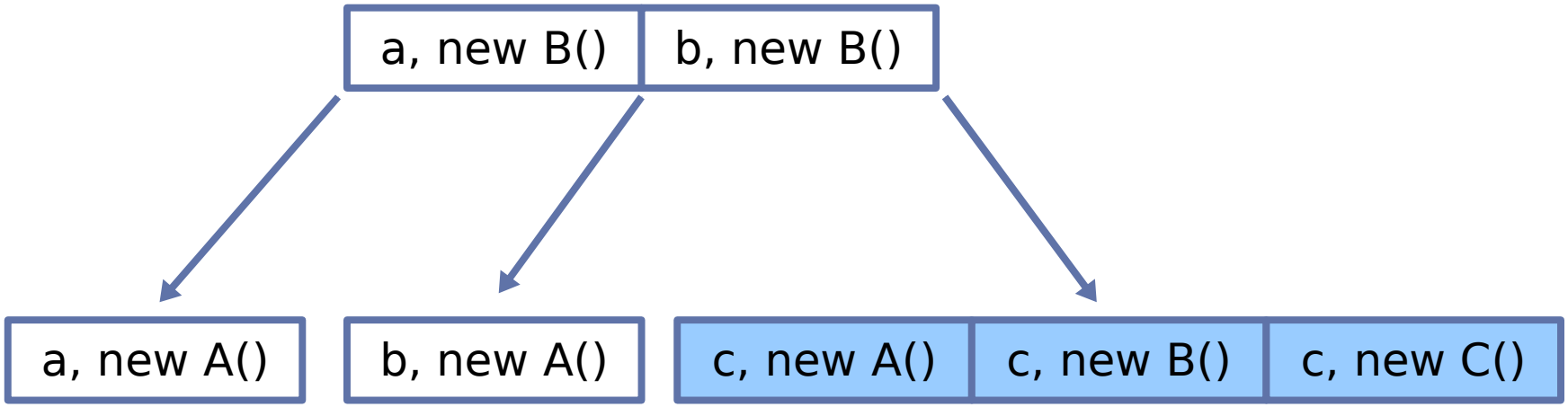
VarPointsTo

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

index organized by
reverse argument order

```
VarPointsTo(?var, ?obj)
```

- `VarPointsTo(?obj, ?var)`



relations as multidimensional indexes

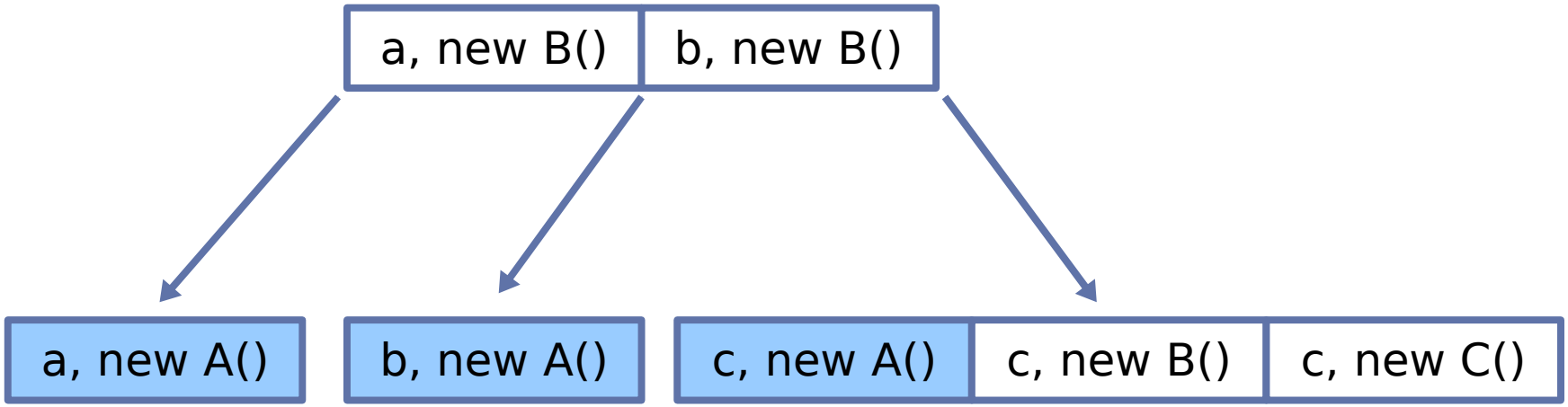
VarPointsTo

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

index organized by
reverse argument order

```
VarPointsTo(?var, ?obj)
```

- ```
  VarPointsTo(?obj, ?var)
  ```

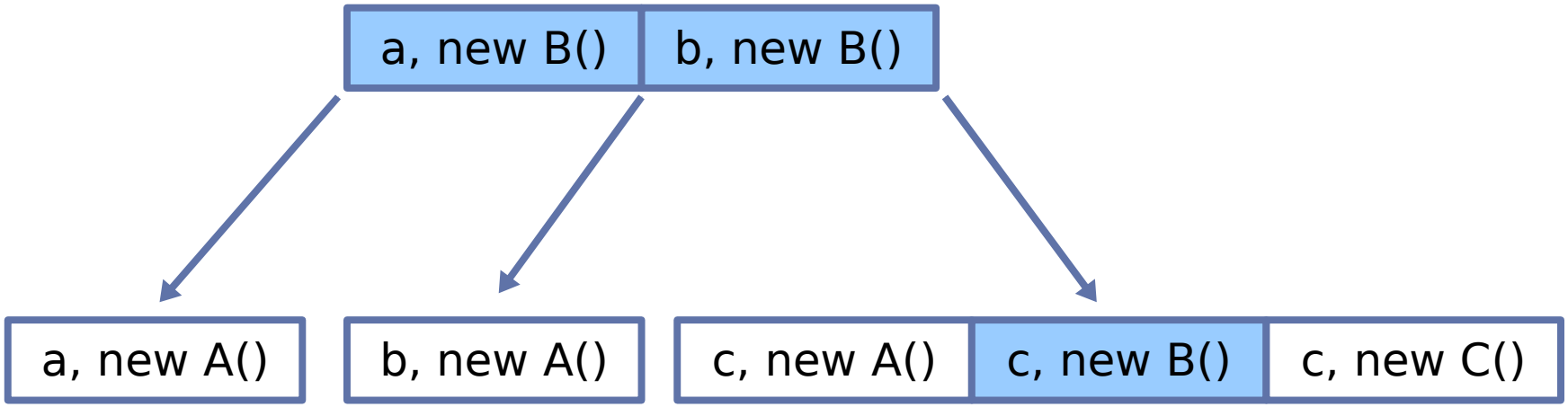

  ```

VarPointsTo

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

index organized by
reverse argument order

- `VarPointsTo(?var, ?obj)`
- `VarPointsTo(?obj, ?var)`



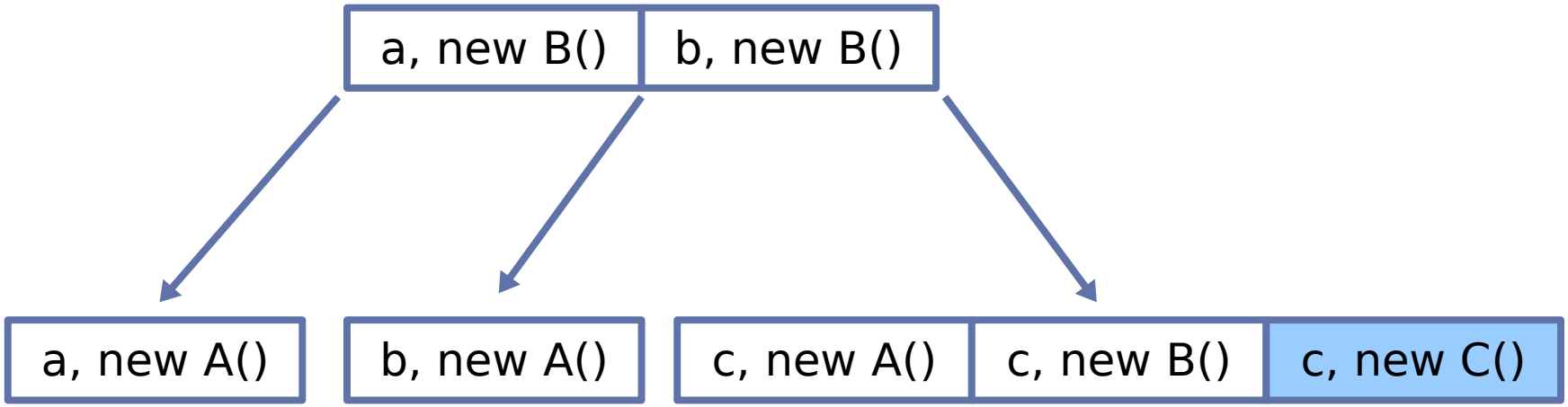
relations as multidimensional indexes

VarPointsTo

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

index organized by
reverse argument order

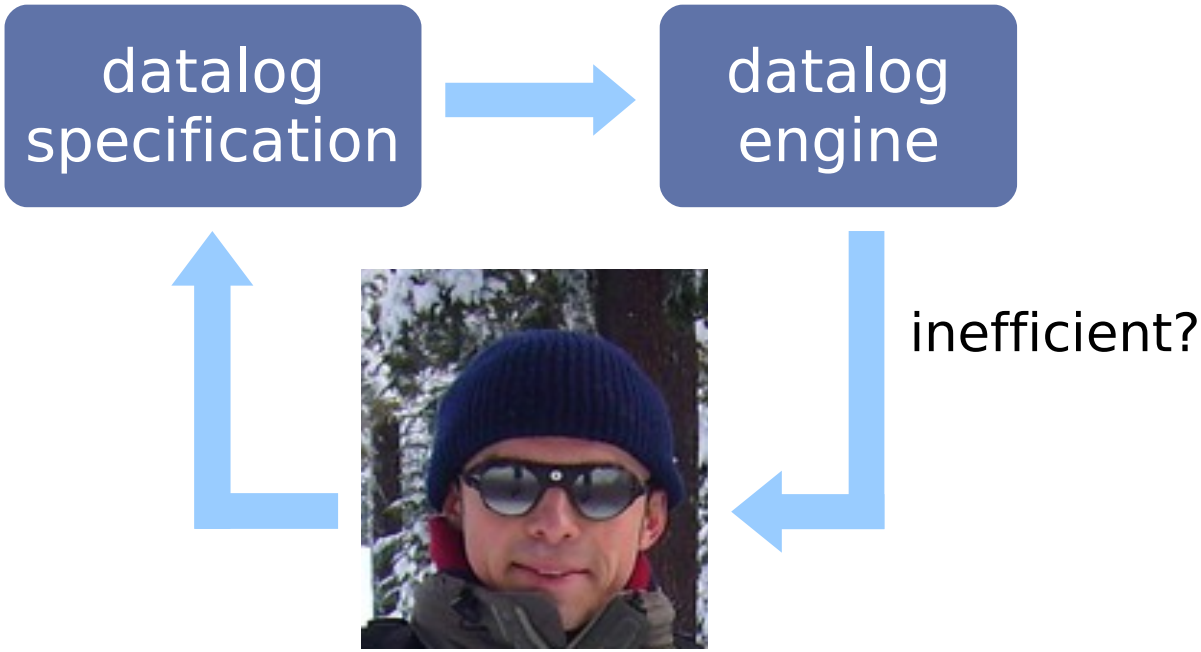
- `VarPointsTo(?var, ?obj)`
- `VarPointsTo(?obj, ?var)`



from specification to algorithms

heuristics for searching the algorithm space

optimization approach: heuristics



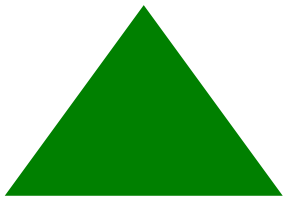
traditional design of single algorithm major human effort!

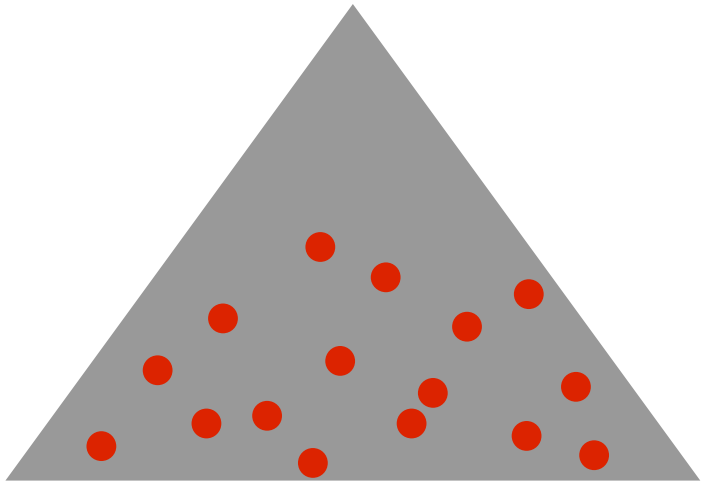
heuristics for searching algorithm space

always use index efficiently

always use index efficiently

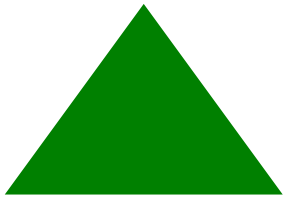
- ΔVarPointsTo(?from, ?obj) $\bowtie$ Assign(?from, ?to)

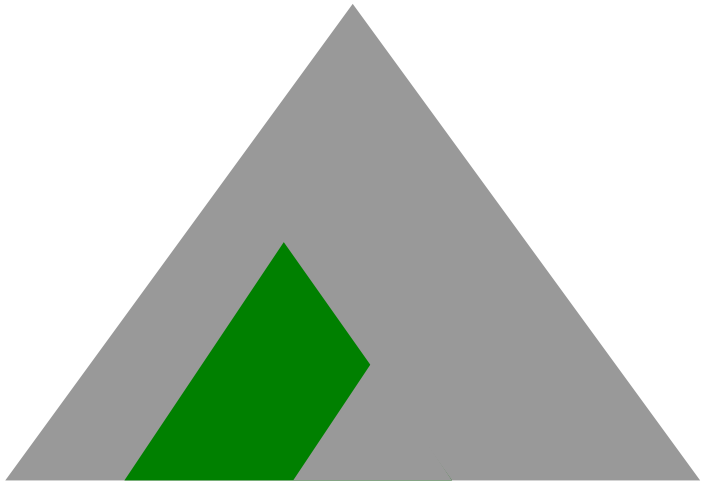




always use index efficiently

- ΔVarPointsTo(?from, ?obj) $\bowtie$ Assign(?from, ?to)
- ΔVarPointsTo(?from, ?obj) $\bowtie$ Assign(?to, ?from)





heuristics for searching algorithm space

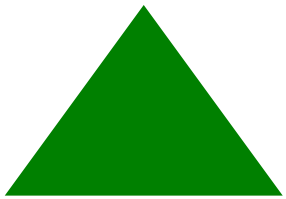
always use index efficiently

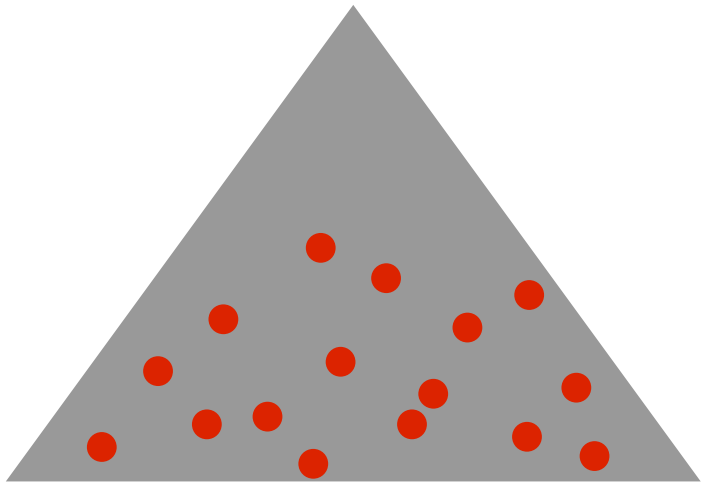
never iterate over full views

always use index efficiently

never iterate over full views

- ΔAssign(?to, ?from) $\bowtie$ VarPointsTo(?from, ?obj)

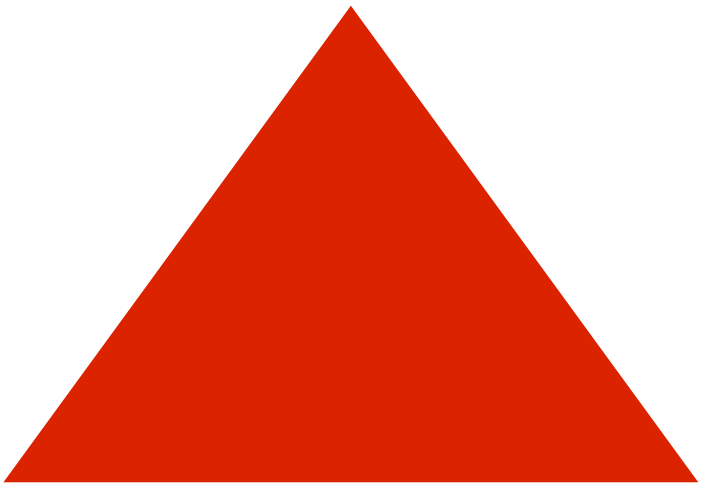


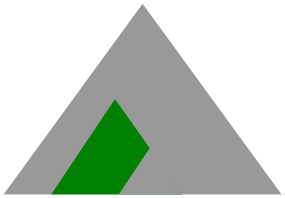


always use index efficiently

never iterate over full views

- ΔAssign(?to, ?from) $\bowtie$ VarPointsTo(?from, ?obj)
- VarPointsTo(?from, ?obj) $\bowtie$ ΔAssign(?to, ?from)



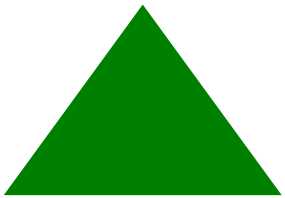


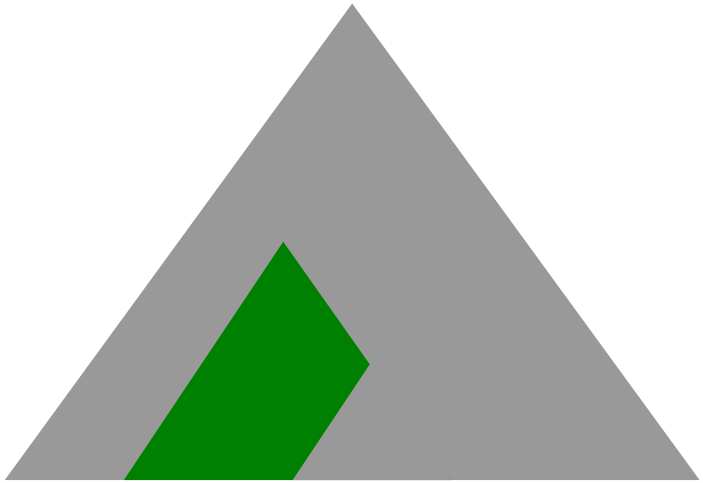
heuristics for searching algorithm space

always use index efficiently

never iterate over full views

- ΔAssign(?to, ?from) ⋈ VarPointsTo(?from, ?obj)
- VarPointsTo(?from, ?obj) ⋈ ΔAssign(?to, ?from)
- ΔAssign(?to, ?from) ⋈ VarPointsTo(?obj, ?from)





heuristics for searching algorithm space

always use index efficiently

never iterate over full views

never iterate over big input relations

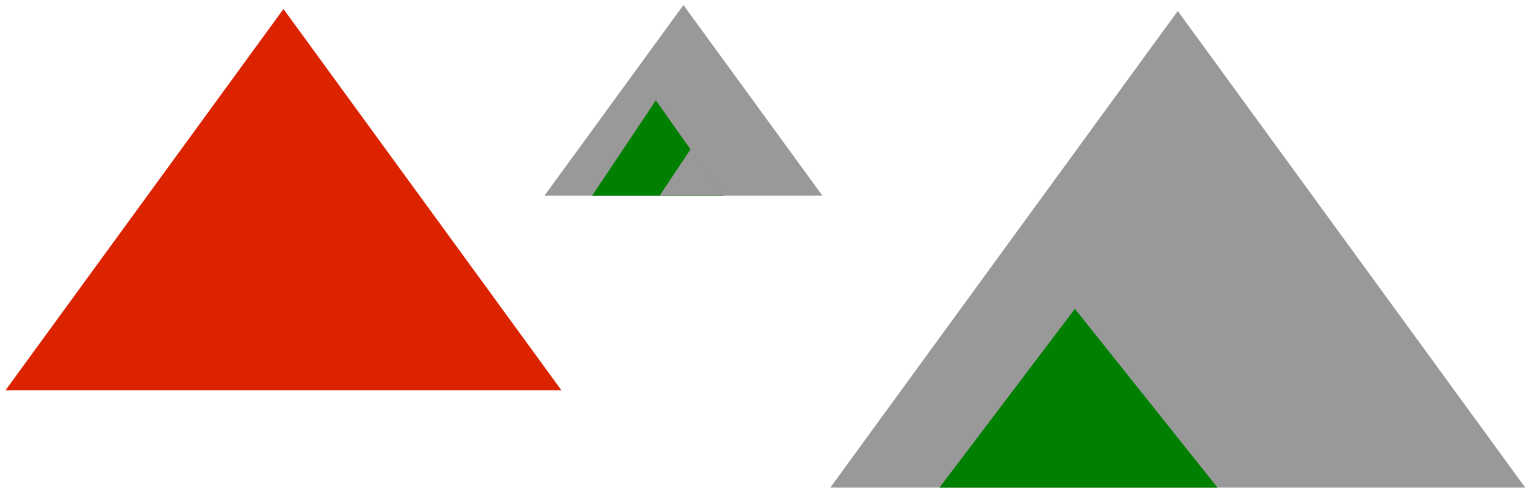
heuristics for searching algorithm space

always use index efficiently

never iterate over full views

never iterate over big input relations

- `StoreField(?from, ?field, ?base)`
  ⋈ `ΔVarPointsTo(?obj, ?from)`
  ⋈ `VarPointsTo(?baseobj, ?base)`



from specification to algorithms

folding transformation example

folding transformation: example

specification

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

specification

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

semi-naive executions

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

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

folding transformation: example

specification

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

join orderings

```
  ΔVarPointsTo(?baseobj, ?base)
  ⋈ StoreField(?from, ?base, ?field)
  ⋈ VarPointsTo(?obj, ?from)

  ΔVarPointsTo(?obj, ?from)
  ⋈ StoreField(?from, ?base, ?field)
  ⋈ VarPointsTo(?baseobj, ?base)
```

folding transformation: example

specification

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

there is no efficient variable ordering!

join orderings

- ΔVarPointsTo(?baseobj, ?base)
  $\bowtie$ StoreField(?from, ?base, ?field)
  $\bowtie$ VarPointsTo(?obj, ?from)

- ΔVarPointsTo(?obj, ?from)
  $\bowtie$ StoreField(?from, ?base, ?field)
  $\bowtie$ VarPointsTo(?baseobj, ?base)

folding transformation: example

specification

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

there is no efficient variable ordering!

join orderings

```
  ΔVarPointsTo(?baseobj, ?base)
  ⋈ StoreField(?from, ?base, ?field)
  ⋈ VarPointsTo(?obj, ?from)

  ΔVarPointsTo(?obj, ?from)
  ⋈ StoreField(?from, ?base, ?field)
  ⋈ VarPointsTo(?baseobj, ?base)
```

specification

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

there is no efficient variable ordering!

join orderings

```
ΔVarPointsTo(?baseobj, ?base)
⋈ StoreField(?from, ?base, ?field)
⋈ VarPointsTo(?obj, ?from)

ΔVarPointsTo(?obj, ?from)
⋈ StoreField(?from, ?base, ?field)
⋈ VarPointsTo(?baseobj, ?base)
```

specification

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

specification

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

optimized rules

```
FieldPointsTo(?obj, ?field, ?baseobj) <-

  VarPointsTo(?obj, ?from).

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

specification

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

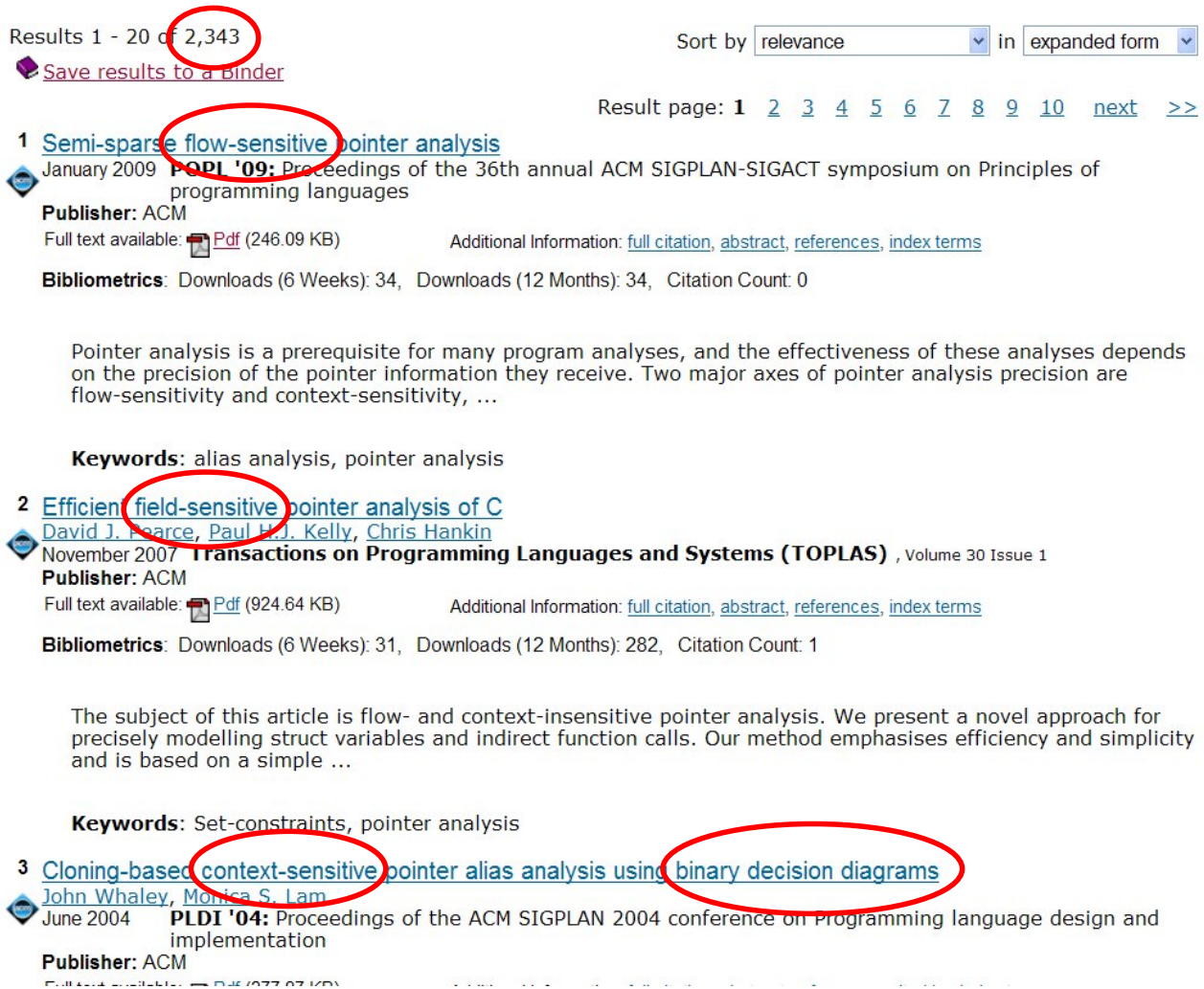
optimized rules

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

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

algorithm space is gigantic

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

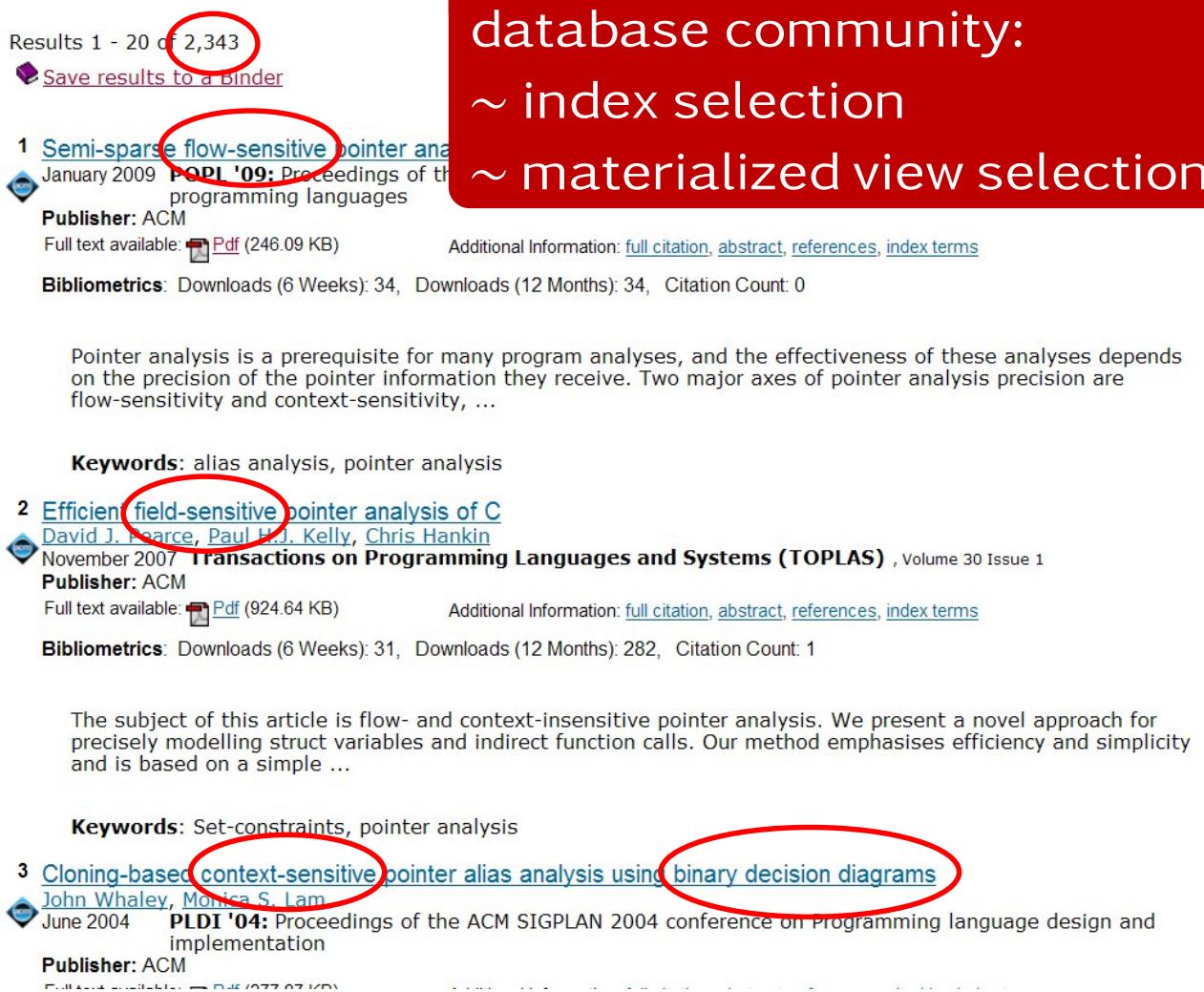
John Whaley, Monica S. Lam

June 2004 **PLDI '04:** Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation

Publisher: ACM

algorithm space is gigantic

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



Results 1 - 20 of 2,343

Save results to a Binder

1 Semi-sparse flow-sensitive pointer ana

January 2009 **POPL '09:** Proceedings of th 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

database community:
~ index selection
~ materialized view selection

from algorithms to specification

from specification to algorithms

fundamental new approach to pointer analysis 33

distilled domain of pointer analysis to its essence

- declarative specification
- fully captured the domain: no loss of sophistication
  - even improved: exception, reflection analysis
  - declarativiness is a major benefit here

developed techniques for deriving efficient algorithms

- effective heuristics for searching the algorithm space
- outperforms current hand-written implementations

  $\Rightarrow$ evidence that this is an effective method!

pointer analysis

- new context abstractions
- selective context-sensitivity
- on-demand analysis
- improve evaluation methods

pointer analysis

- new context abstractions
- selective context-sensitivity
- on-demand analysis
- improve evaluation methods

program analysis and applications

- mixed language analysis
- declarative dynamic program analysis
- applications: client analyses

pointer analysis

- new context abstractions
- selective context-sensitivity
- on-demand analysis
- improve evaluation methods

program analysis and applications

- mixed language analysis
- declarative dynamic program analysis
- applications: client analyses

transformation systems, parsing, extensible languages

- too much to list here!