Software Challenges: Abstraction and Analysis

Martin Bravenboer

University of Massachusetts Amherst
University of Oregon

April 2, 2009

University of Waterloo

the design and implementation of programming language features and software engineering tools for solving software development problems

extensible and domain-specific languages

brief overview

program transformation

no time :′(

program analysis

latest work, rest of the talk

examples of poor abstraction and language integration

example 1: embedding in string literals


```java
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

example 1: embedding in string literals

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

password `ilovewaterloo`

```
SELECT id FROM users
WHERE name = 'martin' AND
      password = 'ilovewaterloo'
```

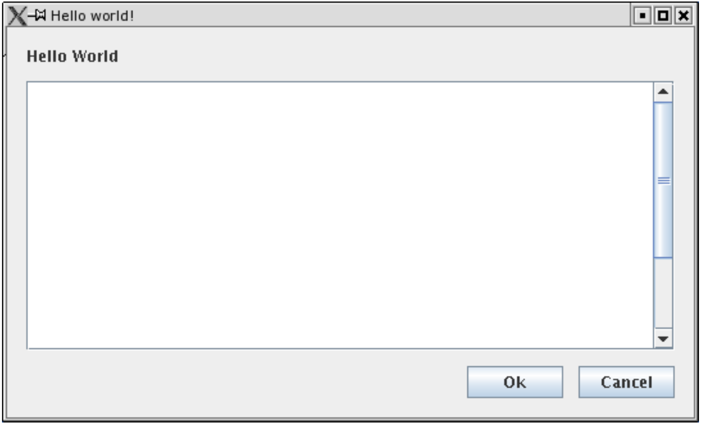
example 1: embedding in string literals

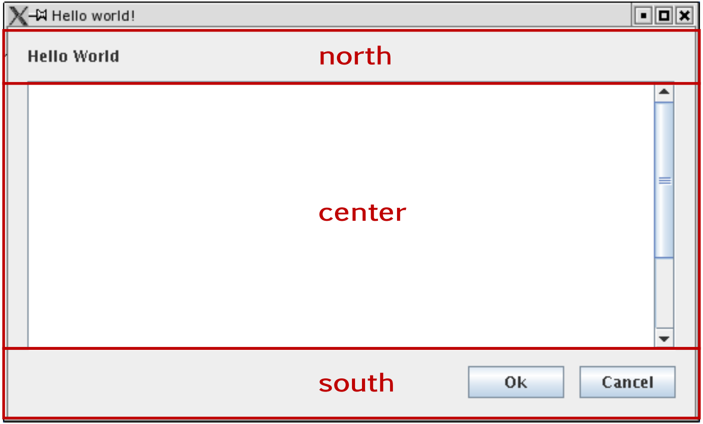
```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

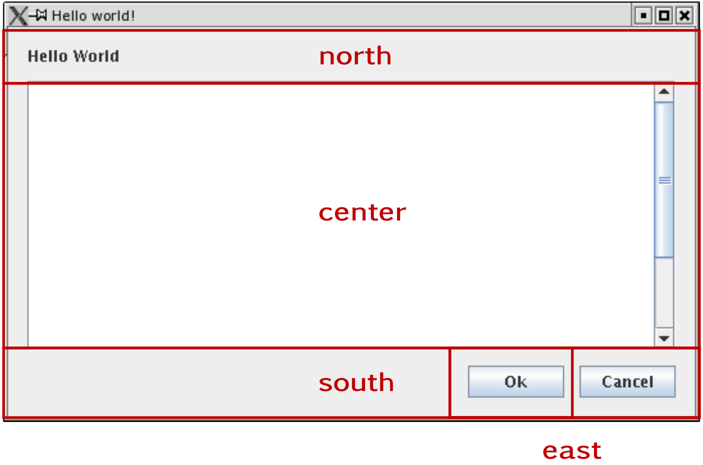
```
password ' OR 'x' = 'x'
```

```
SELECT id FROM users
WHERE name = 'dr.evil' AND
      password = '' OR 'x' = 'x'
```

example 2: implement a graphical user-interface




Hello world!
Hello World
north
center
south
Ok
Cancel


Hello world!
Hello World
north
center
south
Ok
Cancel
east

example 2: implement a graphical user-interface


```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons);
    panel.add(BorderLayout.SOUTH, south);
    ...
```

example 2: implement a graphical user-interface


```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons);
    panel.add(BorderLayout.SOUTH, south);
    ...
```

example 2: implement a graphical user-interface


```
public class HelloWorld {
  public static void main(String[] ps) {

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons
    panel.add(BorderLayout.SOUTH, south)
    ...
```

Does not correspond to hierarchical structure of the user-interface.

example 2: implement a graphical user-interface


```
public class HelloWorld {
  public static void main

    JTextArea text = new JTextArea(20,40);

    JPanel panel = new JPanel(new BorderLayout(12,12));
    panel.add(BorderLayout.NORTH , new JLabel("Hello World"));
    panel.add(BorderLayout.CENTER , new JScrollPane(text));

    JPanel south = new JPanel(new BorderLayout(12,12));
    JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
    buttons.add(new JButton("Ok"));
    buttons.add(new JButton("Cancel"));

    south.add(BorderLayout.EAST, buttons
    panel.add(BorderLayout.SOUTH, south)
    ...
```

General-purpose languages do not give you the means to write coherent sentences over the vocabulary of an API

Does not correspond to hierarchical structure of the user-interface.

example 3: libraries as half-baked languages


```java
public void testBuysWhenPriceEqualsThreshold() {
  mainframe.expects(once())
    .method("buy").with(eq(QUANTITY))
    .will(returnValue(TICKET));

  auditing.expects(once())
    .method("bought").with(same(TICKET));

  agent.onPriceChange(THRESHOLD);
}
```

```java
public void testBuysWhenPriceEqualsThreshold() {
  mainframe.expects(once())
    .method("buy").with(eq(QUANTITY))
    .will(returnValue(TICKET));

  auditing.expects(once())
    .method("bought").with(same(TICKET));

  agent.onPriceChange(THRESHOLD);
}
```

general-purpose languages are not designed for writing coherent sentences in a domain-specific language

example 4: failed effort for specific combinations

```
class Foo {
  pointcut foo() : call(boolean *.if*());
}
```

example 4: failed effort for specific combinations

```
class Foo {
  pointcut foo() : call(boolean *.if*());
}
```

```
$ajc Foo.java
Foo.java:2 [error] Syntax error on token "if", invalid
           allowable token in pointcut or type pattern
```

example 4: failed effort for specific combinations

```
class Foo {
  pointcut foo() : call(boolean *.if*());
}
```

```
$ajc Foo.java
Foo.java:2 [error] Syntax error on token "if", invalid
           allowable token in pointcut or type pattern
```

```
class Foo {
  pointcut foo() :  call(void *0.Ef());
}
```

example 4: failed effort for specific combinations

```
class Foo {
  pointcut foo() : call(boolean *.if*());
}
```

```
$ajc Foo.java
Foo.java:2 [error] Syntax error on token "if", invalid
           allowable token in pointcut or type pattern
```

```
class Foo {
  pointcut foo() :  call(void *0.Ef());
}
```

```
$ajc Foo.java
Foo.java:2 [error] Invalid float literal number
pointcut foo() :  call(void *0.Ef());
                             ^
```

improved abstraction with embedded domain-specific languages

java and sql

```
String q = "SELECT id FROM users "
         + "WHERE name = '" + userName + "'"
         + "AND password = '" + password + "'";
if (executeQuery(q.toString()).size() == 0) ...
```

```
SQL q = <| SELECT id FROM users
           WHERE name = ${userName}
           AND password = ${password} |>;
if (executeQuery(q.toString()).size() == 0) ...
```

php and shell commands

```
$command = "svn cat \"file name\" -r" . $rev;
system($command);
```

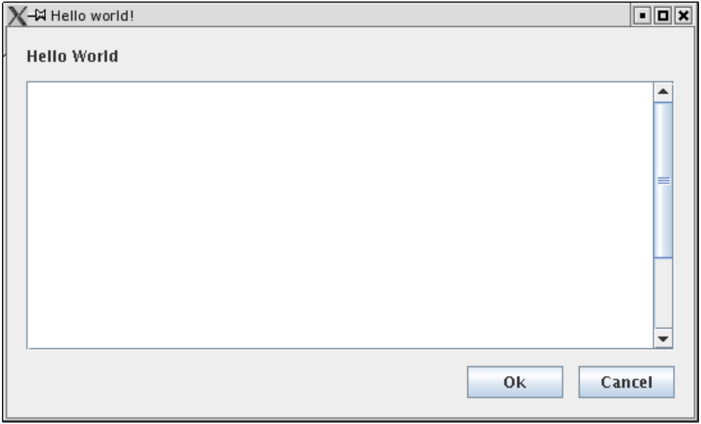
```
$command = <| svn cat "file name" -r${$rev} |>;
system($command->toString());
```

java and ldap

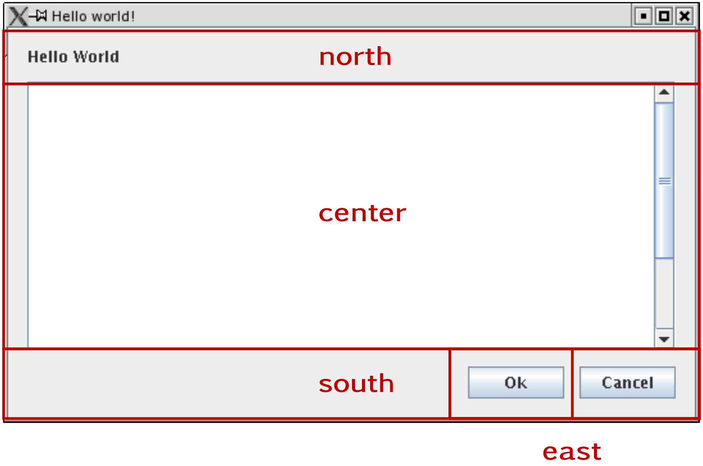
```
String q = "(cn=" + name + ")";
```

```
LDAP q = (| (cn=$(name)) |);
```

example 2: implement a graphical user-interface 10



example 2: implement a graphical user-interface 10



example 2: implement a graphical user-interface 11

```
public static void main(String[] ps) {
  JPanel panel = panel of border layout {
    north = label "Hello World"

    center = scrollpane of textarea {
      rows    = 20
      columns = 40
    }

    south = panel of border layout {
      east = panel of grid layout {
        row = {
          button "Ok"
          button "Cancel"
        }
      }
    }
  }; ...
```

example 2: implement a graphical user-interface 11

```
public static void main(String[] ps) {
  JPanel panel = panel of border layout {
    north = label "Hello World"

    center = scrollpane of textarea {
      rows    = 20
      columns = 40
    }

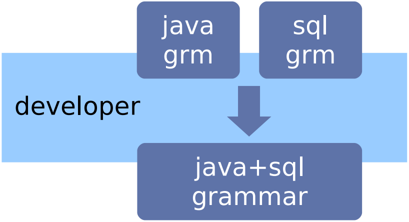
    south = panel of border layout {
      east = panel of grid layout {
        row = {
          button "Ok"
          button "Cancel"
        }
      }
    }
  }; ...
```

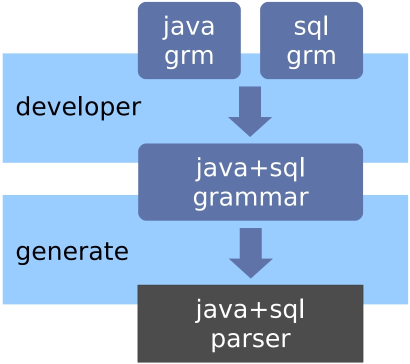
Syntax reflects the hierarchical structure of the user-interface

The interaction between the domain-specific and general-purpose code is seamless we need to fundamentally change the way we develop grammars java grm

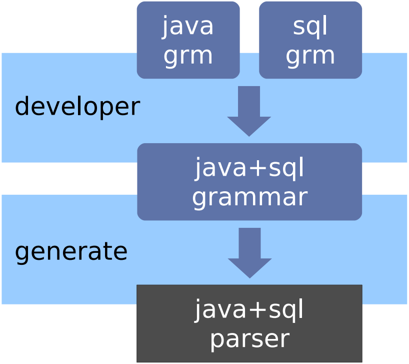
sql grm

technical challenge: grammar composition



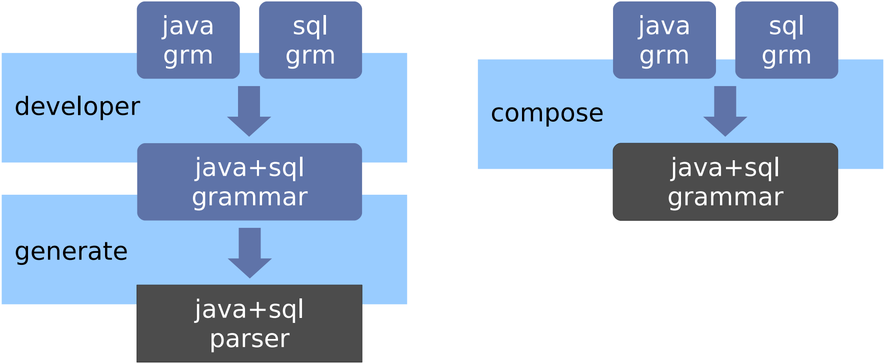

java
grm
sql
grm
developer
java+sql
grammar
generate
java+sql
parser

technical challenge: grammar composition

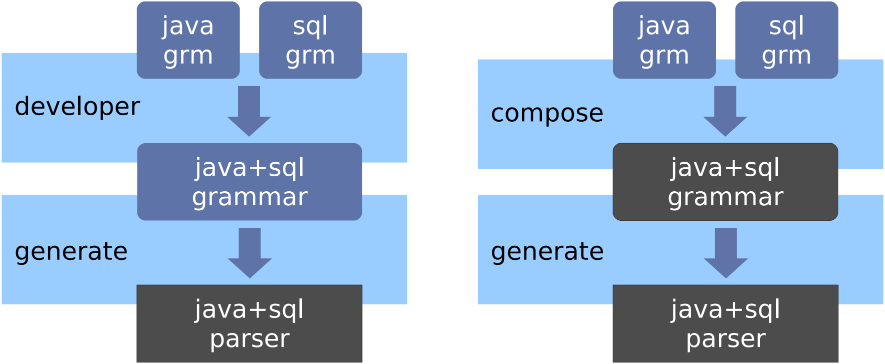


scanners do not compose
→ complex stateful scanners and/or bugs

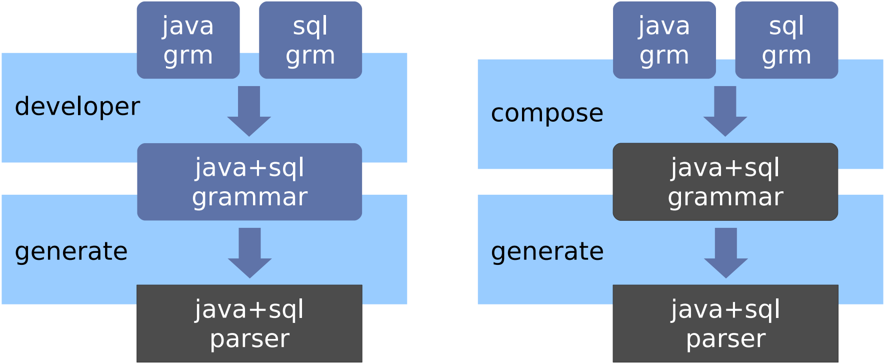
lr grammars do not compose
→ rewrite grammar to resolve conflicts


java
grm
sql
grm
developer
java+sql
grammar
generate
java+sql
parser
java
grm
sql
grm
compose
java+sql
grammar

technical challenge: grammar composition



technical challenge: grammar composition

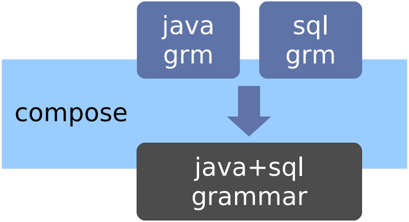


- scanners do not compose → scannerless parser
- lr grammars do not compose → cfg, generalized lr

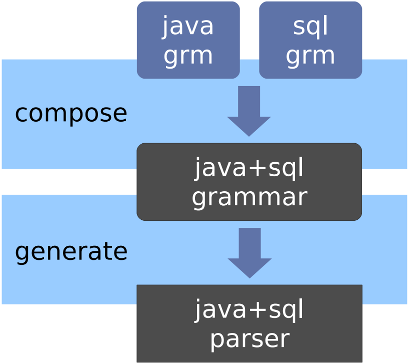
technical challenge: parse table composition

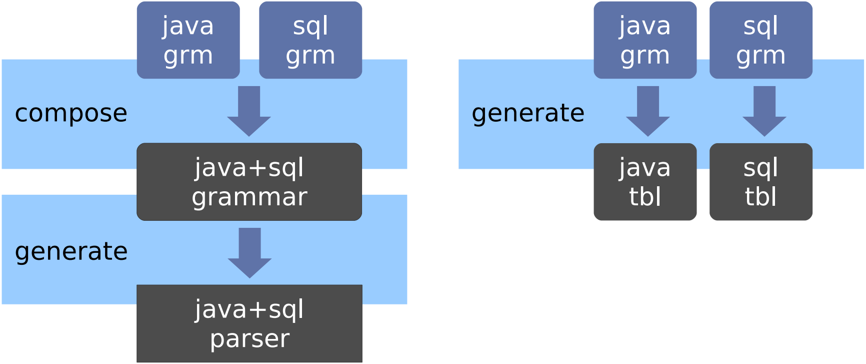
java grm

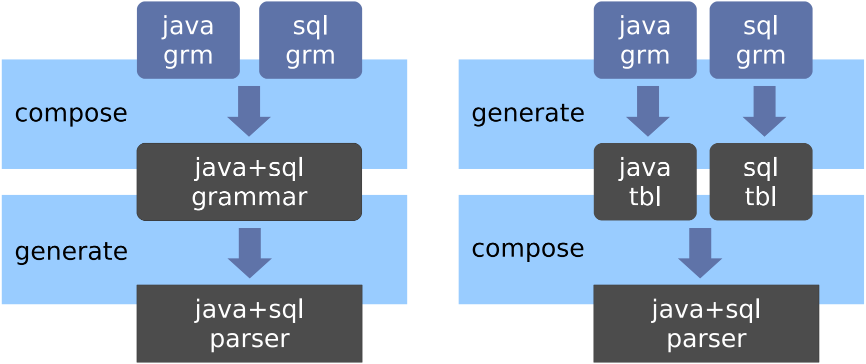
sql grm


java
grm
sql
grm
compose
java+sql
grammar

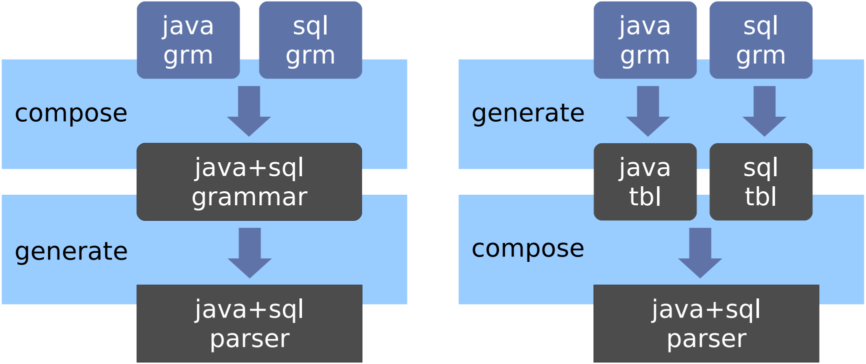
technical challenge: parse table composition




java
grm
sql
grm
compose
java+sql
grammar
generate
java+sql
parser
java
grm
sql
grm
generate
java
tbl
sql
tbl


java
grm
sql
grm
compose
java+sql
grammar
generate
java+sql
parser
java
grm
sql
grm
generate
java
tbl
sql
tbl
compose
java+sql
parser

technical challenge: parse table composition



- from source-level extensibility to binary extensibility
- algorithm: minimize DFA reconstruction

extensible and domain-specific languages

program transformation

program analysis

extensible and domain-specific languages

embedding domain-specific languages

[OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]

>100 citations

best paper

program transformation

program analysis

extensible and domain-specific languages

embedding domain-specific languages
[OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]

preventing injection attacks [GPCE 2007, SCP 2009]

program transformation

program analysis

extensible and domain-specific languages

embedding domain-specific languages
[OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]
preventing injection attacks [GPCE 2007, SCP 2009]
parse table composition [SLE 2008]

program transformation

program analysis

extensible and domain-specific languages

embedding domain-specific languages [OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]

preventing injection attacks [GPCE 2007, SCP 2009]

parse table composition [SLE 2008]

program transformation

stratego/xt transformation system [PEPM 2006, FI 2006, SCP 2008]

program analysis

extensible and domain-specific languages

embedding domain-specific languages
[OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]
preventing injection attacks [GPCE 2007, SCP 2009]
parse table composition [SLE 2008]

program transformation

stratego/xt transformation system [PEPM 2006, FI 2006, SCP 2008]
grammar engineering [LDTA 2007]

program analysis

my research

extensible and domain-specific languages

embedding domain-specific languages
[OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]
preventing injection attacks [GPCE 2007, SCP 2009]
parse table composition [SLE 2008]

program transformation

stratego/xt transformation system [PEPM 2006, FI 2006, SCP 2008]
grammar engineering [LDTA 2007]

program analysis

scalable declarative pointer analysis [ISSTA 2009]

latest and greatest work, rest of the talk

what do we do?

- fully declarative pointer analysis
- fast, really fast

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

what do we do?

- fully declarative pointer analysis
- fast, really fast

how do we do it?

- novel, aggressive optimization
- exposition of indexes

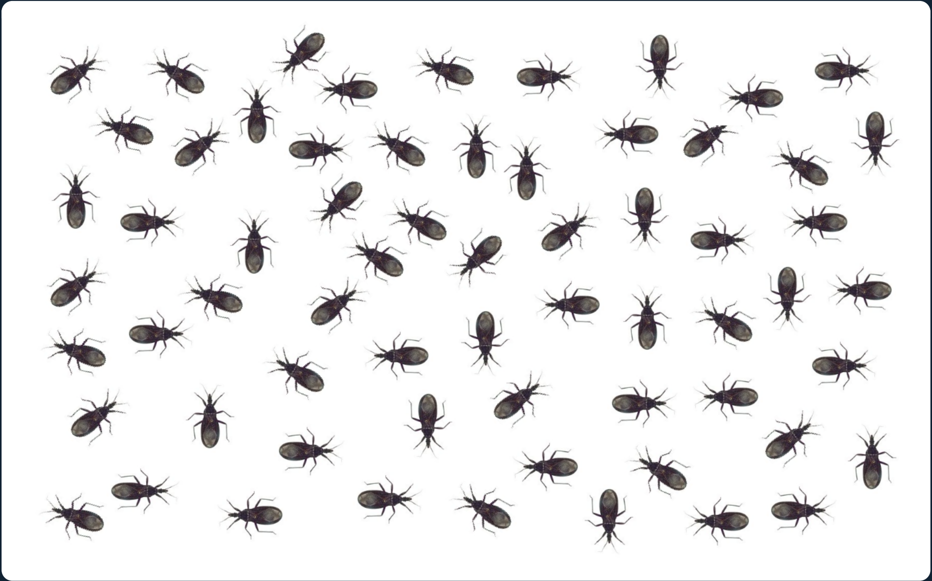
why do you care?

- fast
- sophisticated, simple
- different

why is it relevant?

- optimization
- understanding, find bugs


NO
UNDER-
STANDING
ANY
TIME



what objects can a variable point to?

program

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

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

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

what objects can a variable point to?

program

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

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

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

points-to

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

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

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

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

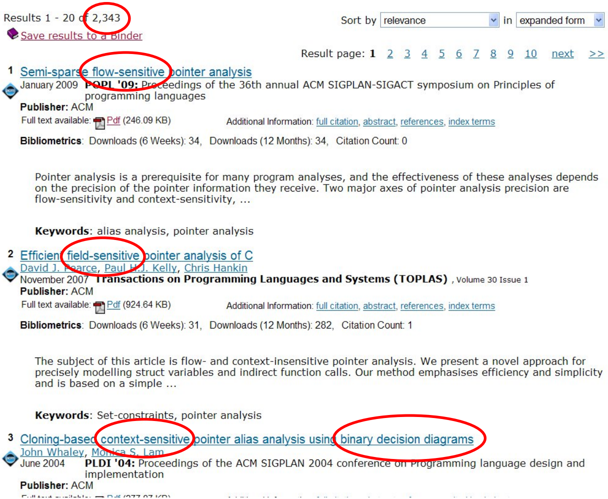
pointer analysis: a complex domain

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

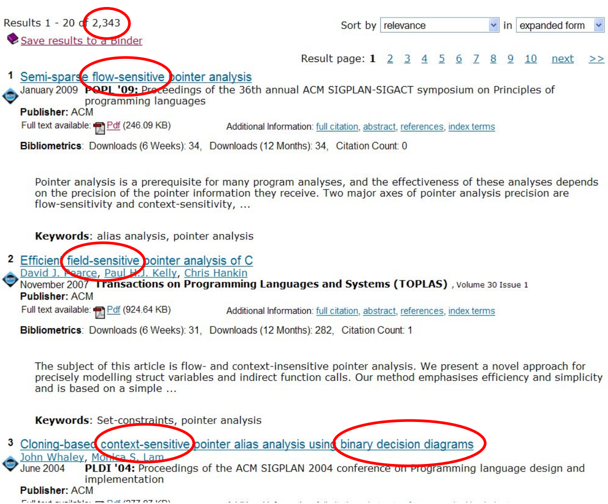
The subject of this article is flow- and context-insensitive pointer analysis. We present a novel approach for precisely modelling struct variables and indirect function calls. Our method emphasises efficiency and simplicity and is based on a simple ...

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

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

```
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*

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

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

procedure *exhaustive_aliasing*(G)
    G:
begin
    1.
    2.
    3.
            $alias\_at\_call\_implies(N, AA, PA, YES)$;
        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

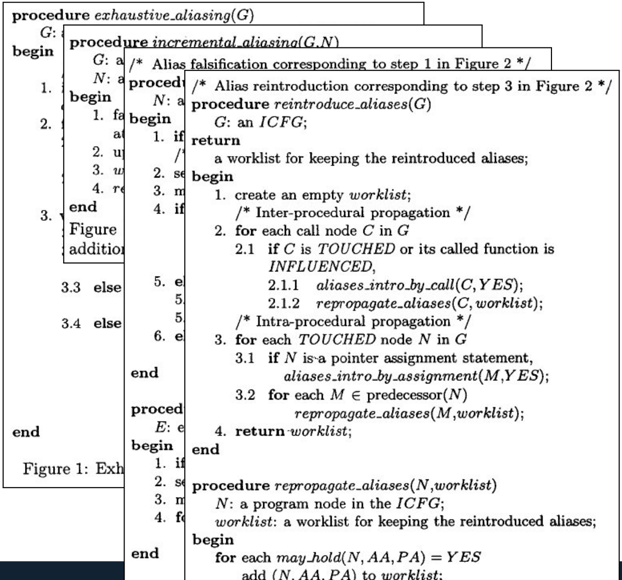
Figure 1: Exhaustive algorithm for *pointer aliasing*

procedure *exhaustive_aliasing*(G)

procedure *incremental_aliasing*(G,N)

```
/* 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
```



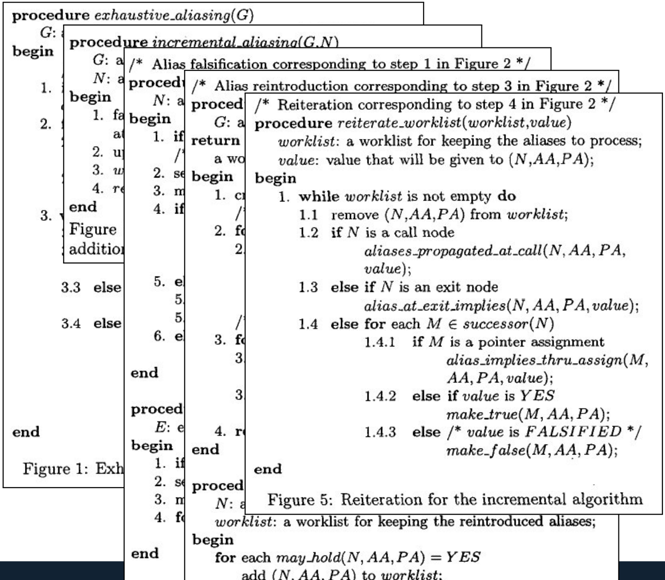
procedure *exhaustive_aliasing*(G)

procedure *incremental_aliasing*(G,N)

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

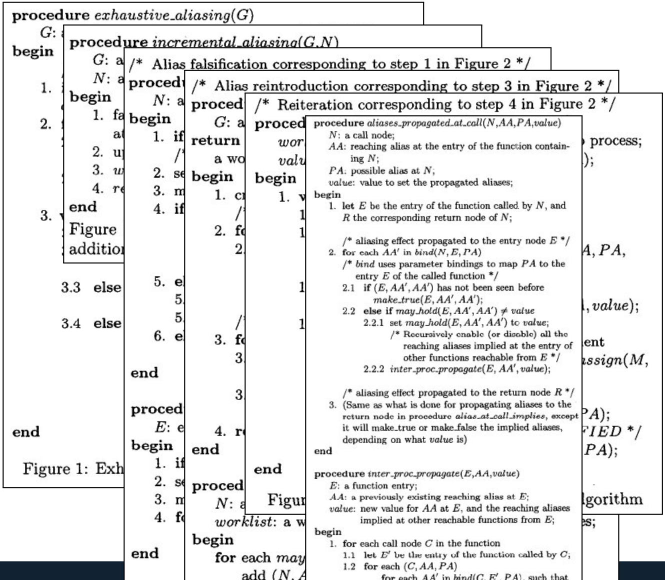
/* Alias reintroduction corresponding to step 3 in Figure 2 */
procedure *reintroduce_aliases*(G)
  G: an *ICFG*;
return
  a worklist for keeping the reintroduced aliases;
begin
  1. create an empty *worklist*;
    /* Inter-procedural propagation */
  2. **for** each call node C in G
    2.1 **if** C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

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

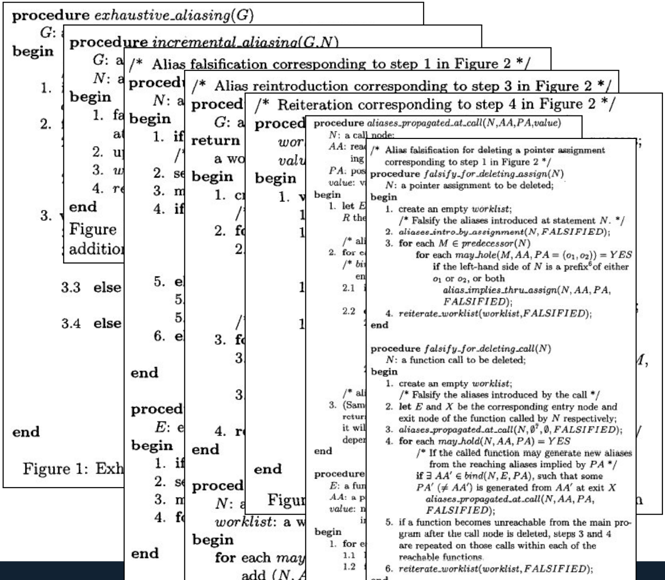


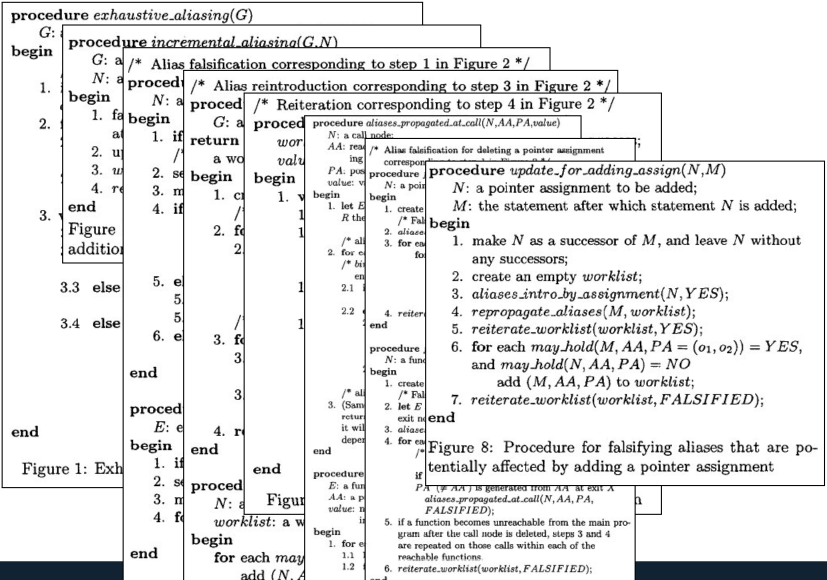
/* Reiteration corresponding to step 4 in Figure 2 */
procedure *reiterate_worklist*(*worklist*,*value*)
worklist: a worklist for keeping the aliases to process;
value: value that will be given to (*N*,*AA*,*PA*);
begin
1. **while** *worklist* is not empty **do**
   1.1 remove (*N*,*AA*,*PA*) from *worklist*;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

Figure 5: Reiteration for the incremental algorithm


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

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 = (o1, o2)) = YES, and may_hold(N, AA, PA) = NO
add (M, AA, PA) to worklist;
7. reiterate_worklist(worklist, FALSIFIED);
end
Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment

algorithms in a 10-page pointer analysis paper

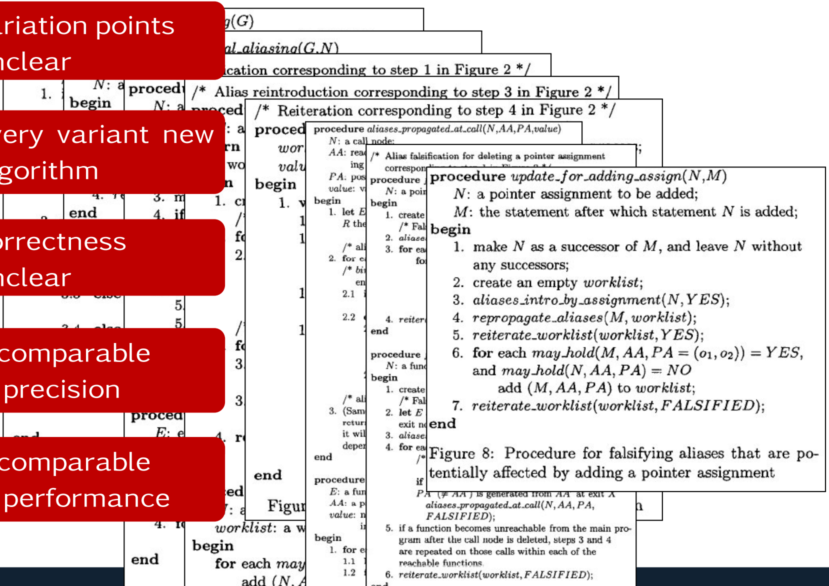
variation points unclear

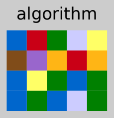
every variant new algorithm

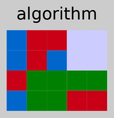
correctness unclear

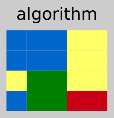
incomparable in precision

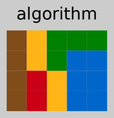
incomparable in performance

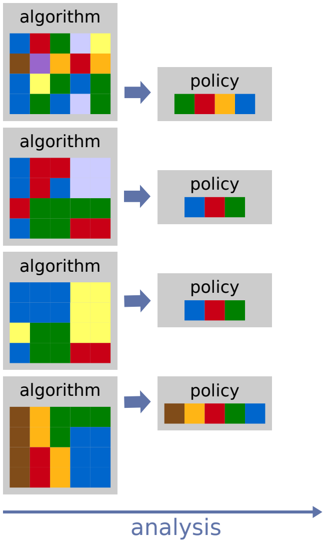



algorithm

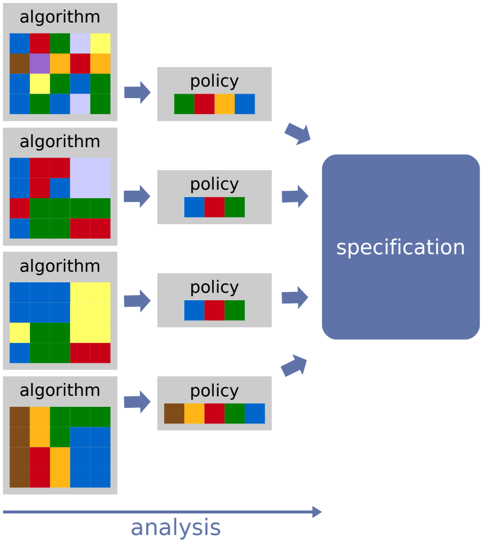

algorithm


algorithm

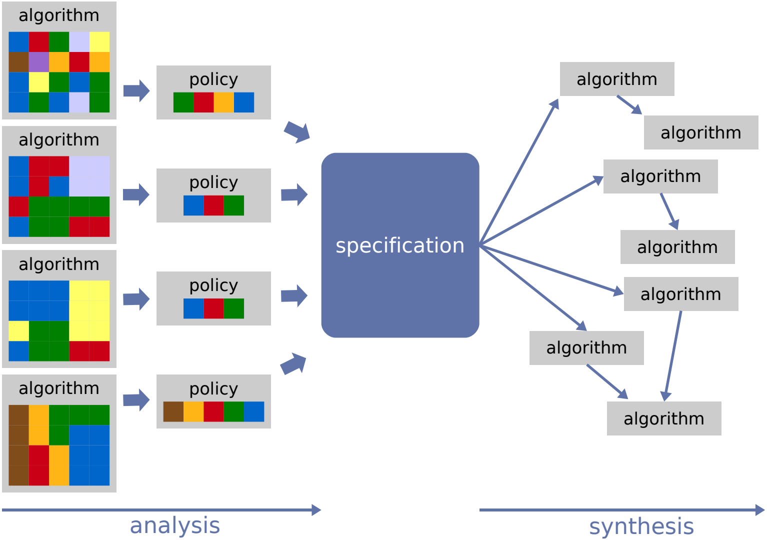

algorithm


algorithm
policy
algorithm
policy
algorithm
policy
algorithm
policy
analysis

from algorithms to specification and back



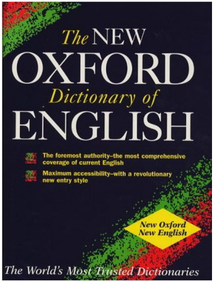
from algorithms to specification and back



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

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

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

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

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

from algorithms to specification

pointer analysis and datalog background

var points-to

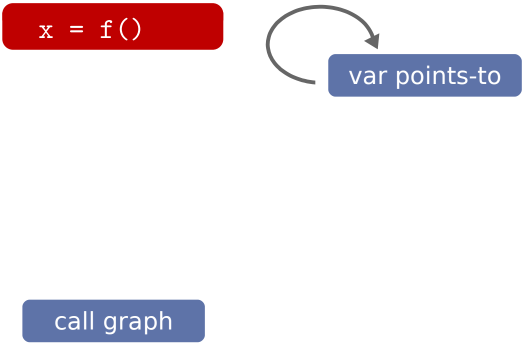
```
x = y
```

var points-to

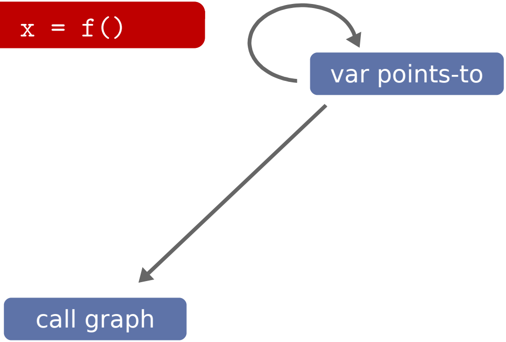
program analysis: a domain of mutual recursion

```
x = y
```

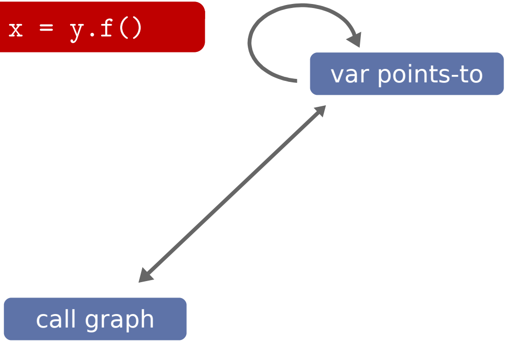
var points-to

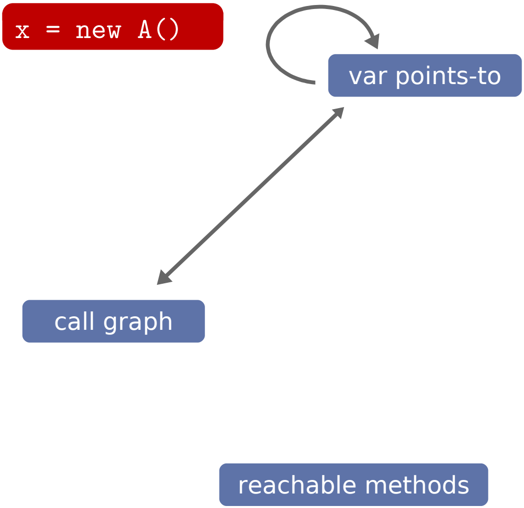

x = f()
var points-to
call graph

```
x = f()
```



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

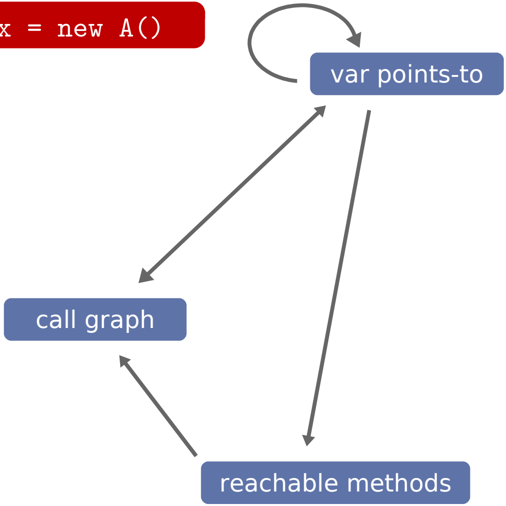


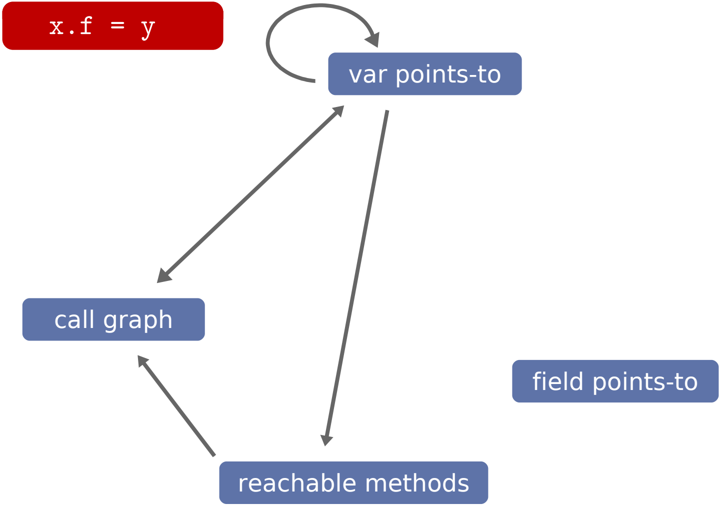


x = new A()
var points-to
call graph
reachable methods

program analysis: a domain of mutual recursion

```
x = new A()
```

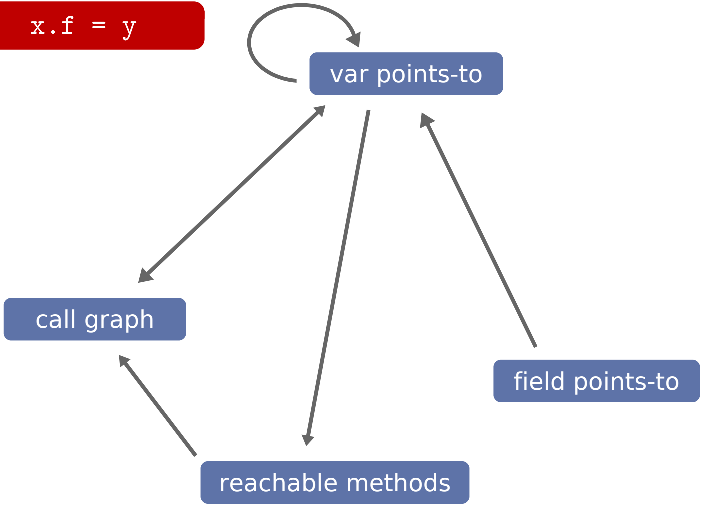




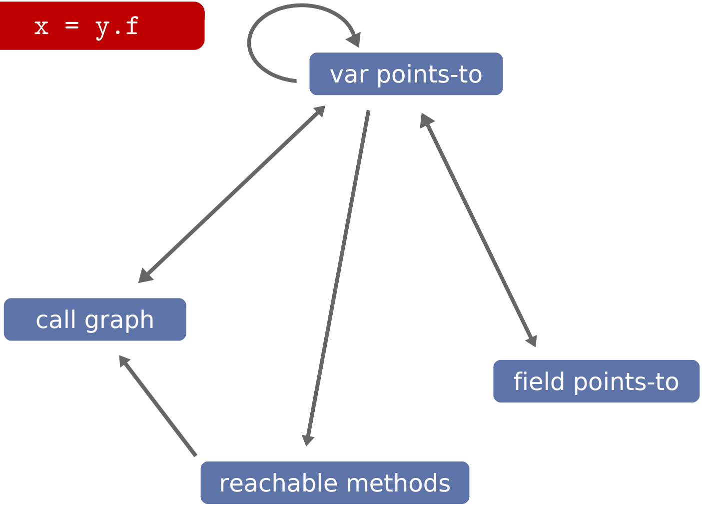
x.f = y
var points-to
call graph
field points-to
reachable methods

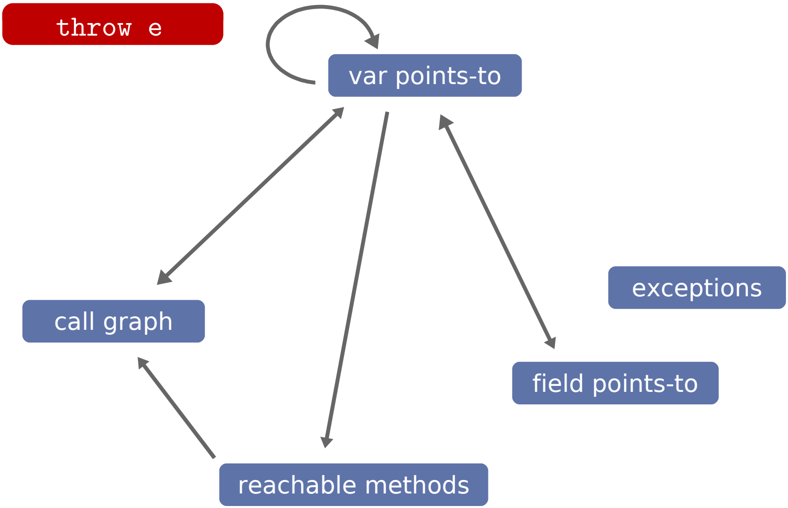
program analysis: a domain of mutual recursion

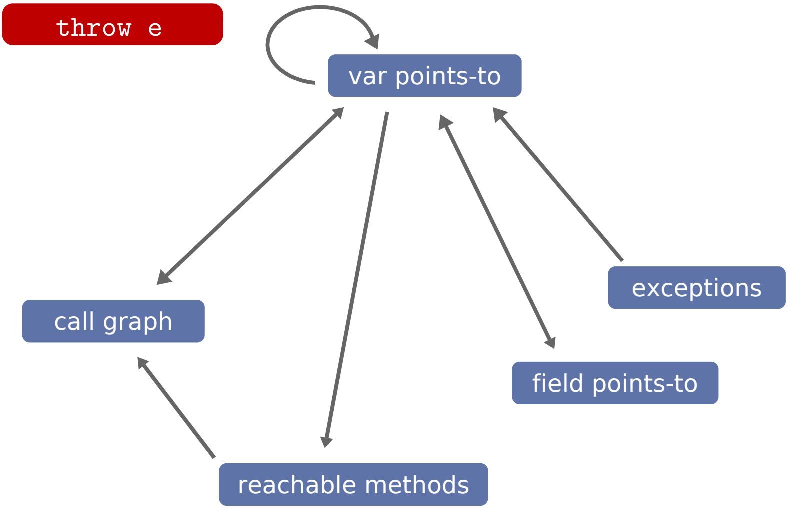
```
x.f = y
```

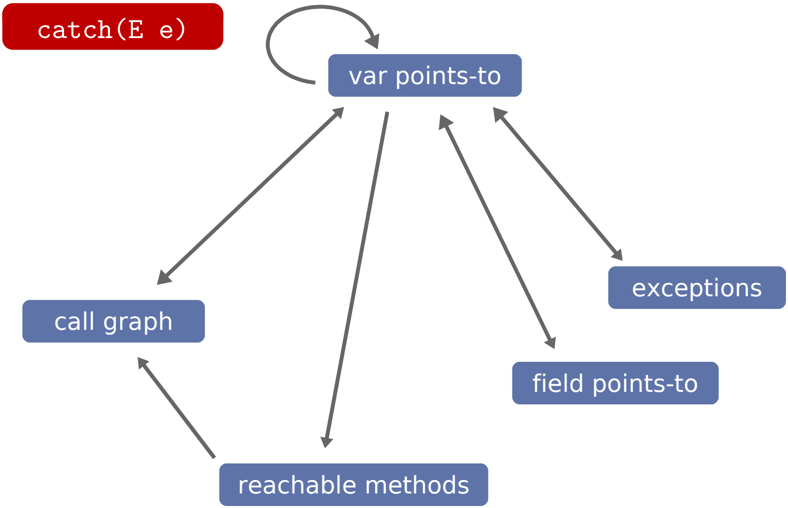


```
x = y.f
```

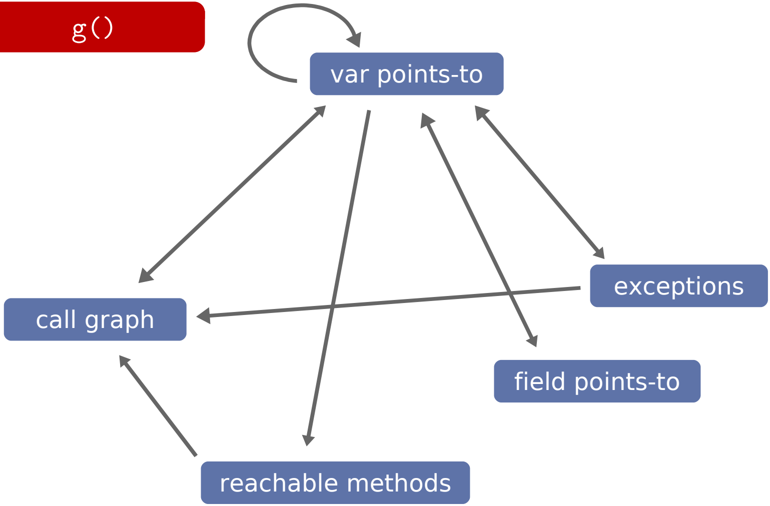



throw e
var points-to
call graph
exceptions
field points-to
reachable methods


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

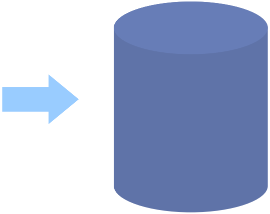

catch(E e)
var points-to
call graph
exceptions
field points-to
reachable methods

`g()`


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

source

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



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).
```

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).
```

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).
```

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).
```

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

example 1: introducing fields


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

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

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

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

?base . ?field = ?from

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

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

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

?base . ?field = ?from

```
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

```
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

```
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

```
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),
```

?baseobj

↑

?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),
  FieldPointsTo(?baseobj, ?field, ?obj).
```

?baseobj

?to = ?base . ?field

example 1: introducing fields

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

```
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).
```

method invocations: propagated exceptions

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

method invocations: caught exceptions

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

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?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),
```

```
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) {..}
}
```

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,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

```
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();
}
```

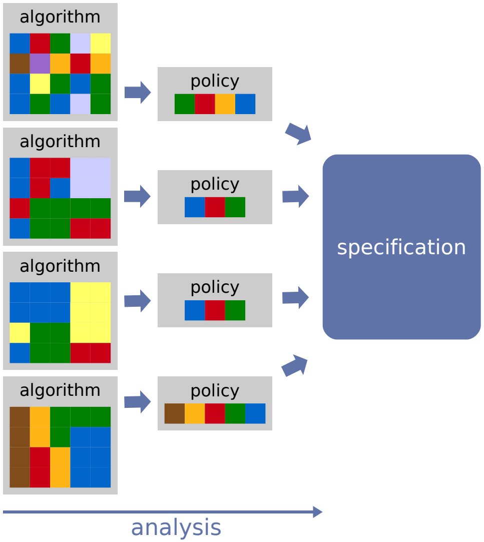
method invocations: caught exceptions

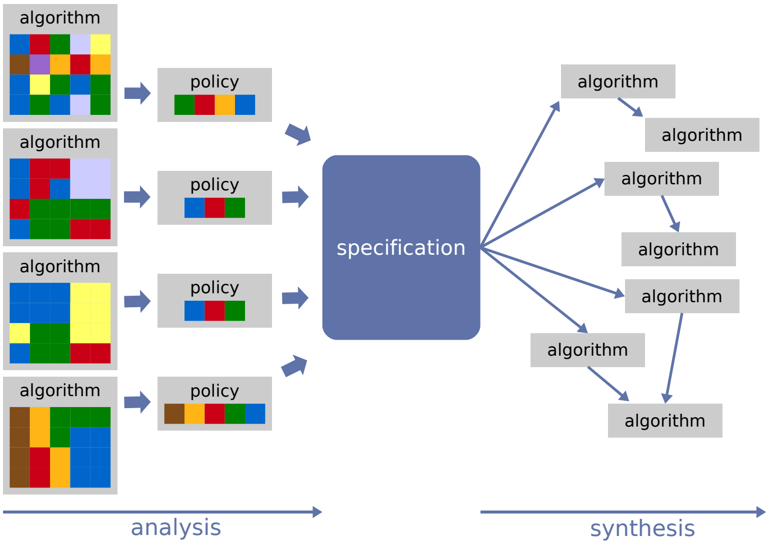
```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  ...
  ...ype, ?invocation] = ?handler,
  ...Param[?handler] = ?param.
```

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

again, new mutually recursive dependencies do not change existing rules

from algorithms to specification and back



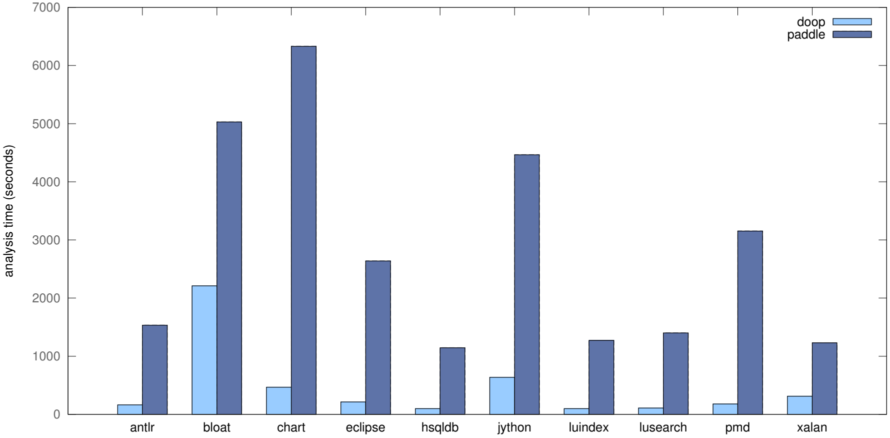

algorithm
algorithm
algorithm
algorithm
policy
policy
policy
policy
specification
algorithm
algorithm
algorithm
algorithm
algorithm
algorithm
algorithm
analysis
synthesis

from specification to algorithms

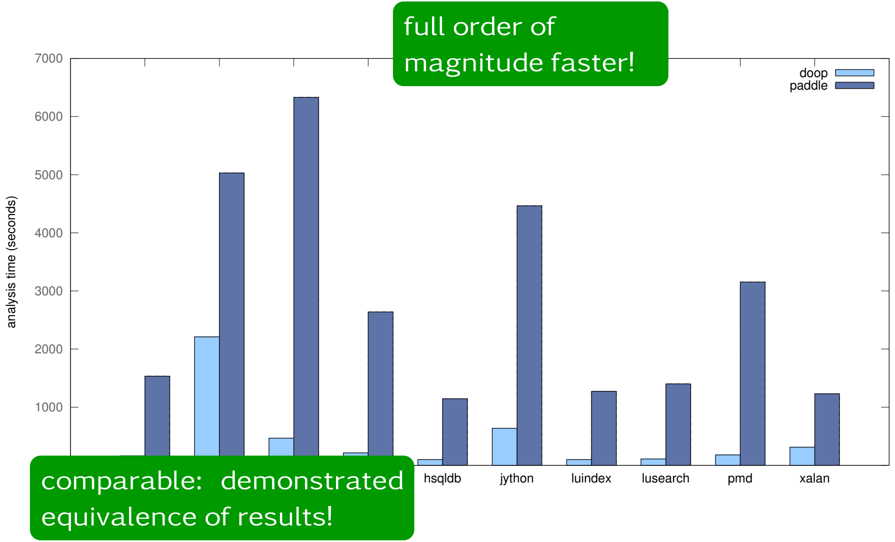
from specification to algorithms

benchmark

benchmark: 1-call-site-sensitive+heap



benchmark: 1-call-site-sensitive+heap



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

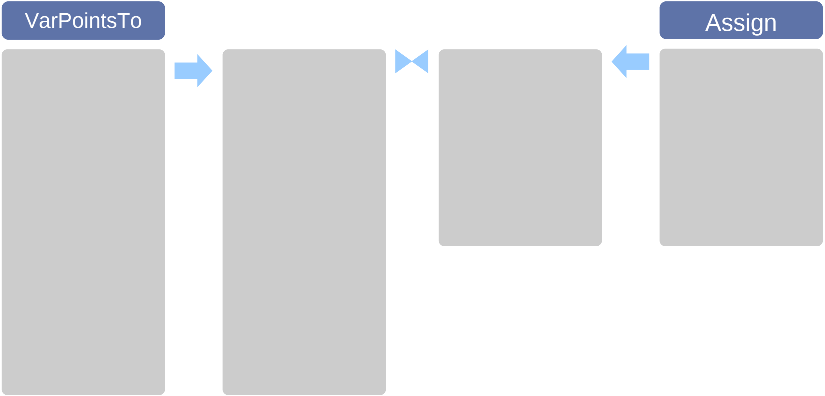
```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?obj, ?var).

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

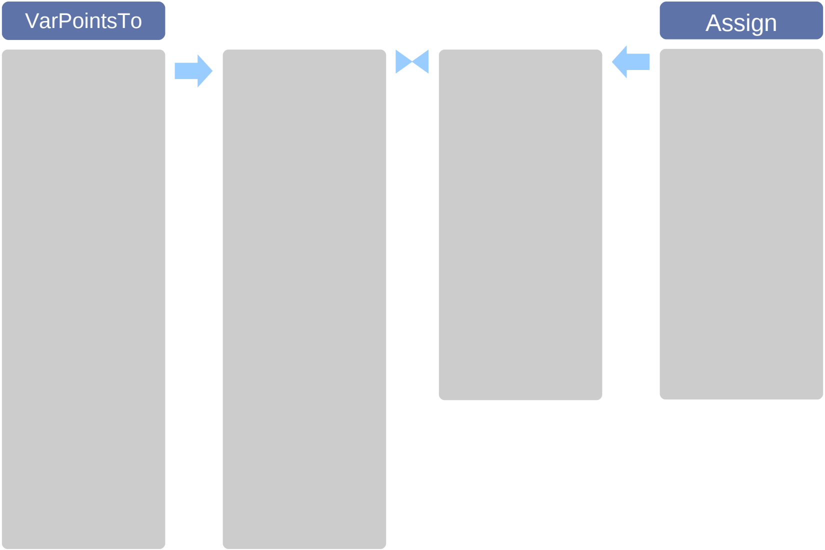
naive evaluation (relational algebra)

```
VarPointsTo := AssignObjectAllocation
repeat
  tmp := π_{to→var,heap}(VarPointsTo ⋈_{from=var} Assign)
  VarPointsTo := VarPointsTo ∪ tmp
until fixpoint
```

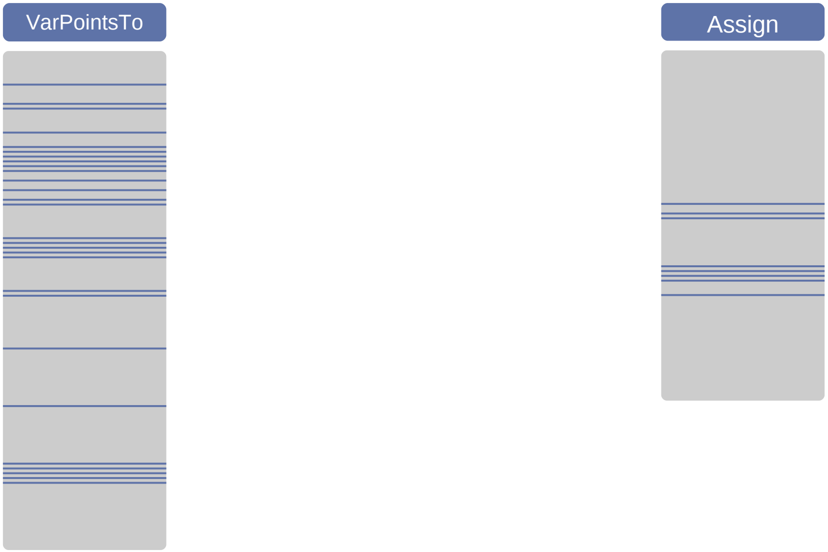
datalog: naive evaluation



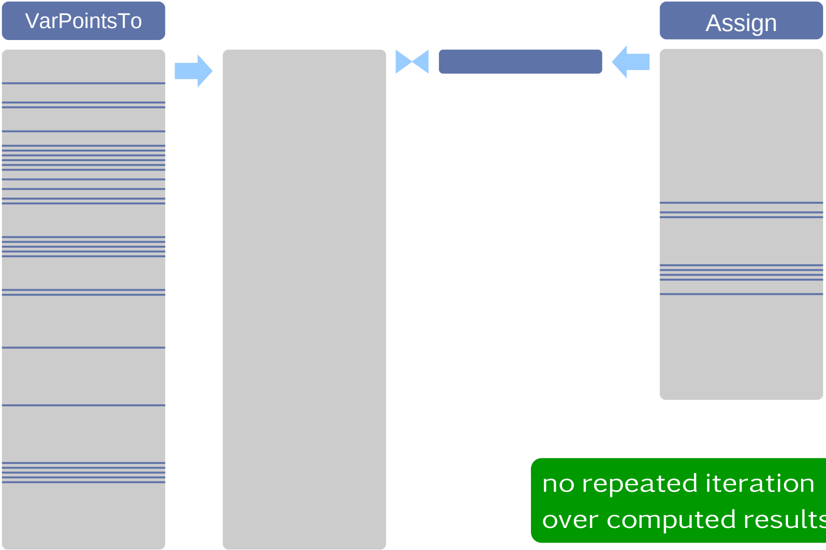
datalog: naive evaluation



datalog: semi-naive evaluation

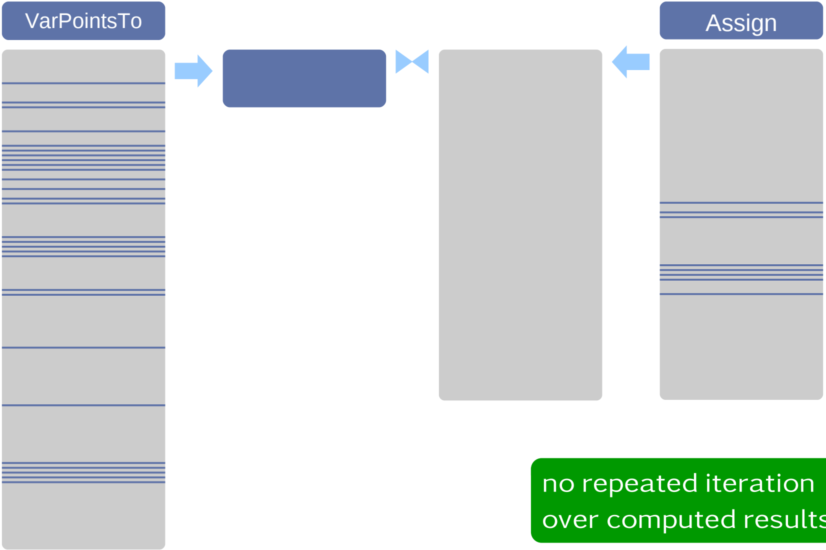


datalog: semi-naive evaluation



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

VarPointsTo

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

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

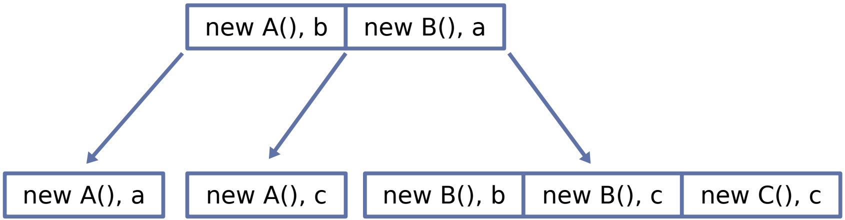
index organized by
reverse argument order

```
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

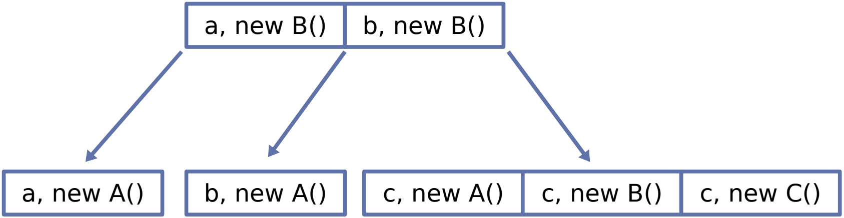
VarPointsTo

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)`

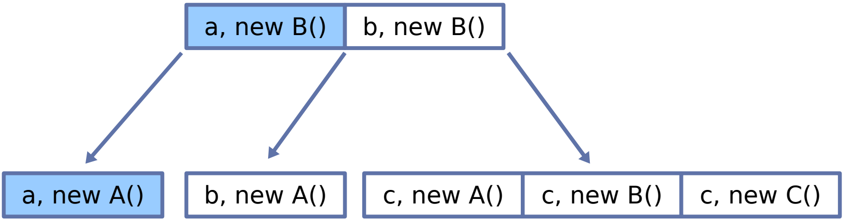


VarPointsTo	
new A()	a
new B()	b
new C()	c
new B()	a
new A()	b
new B()	c
new A()	c

index organized by
reverse argument order

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

- `VarPointsTo(?obj, ?var)`



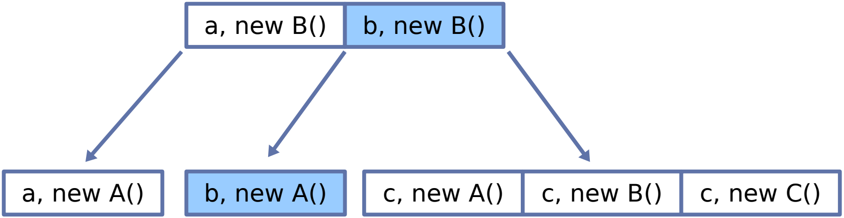
relations as multidimensional indexes

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)`



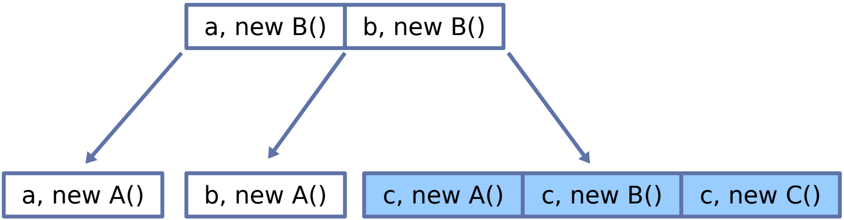
`VarPointsTo`

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)`



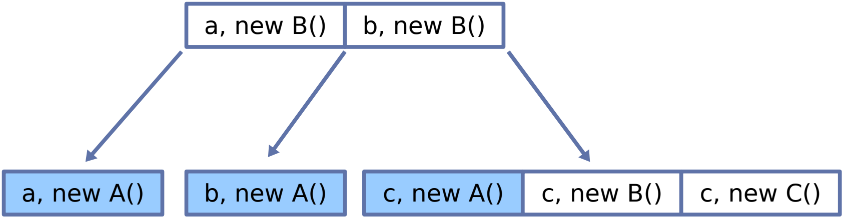
VarPointsTo

`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)
```

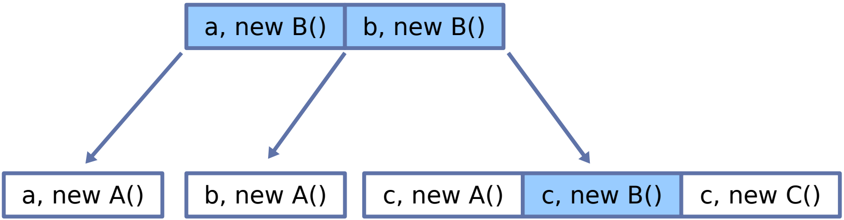


VarPointsTo	
new A()	a
new B()	b
new C()	c
new B()	a
new A()	b
new B()	c
new A()	c

index organized by
reverse argument order

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

- `VarPointsTo(?obj, ?var)`



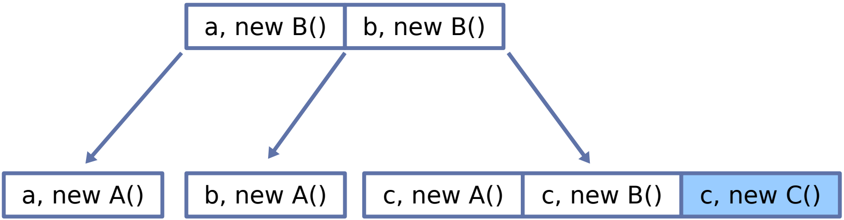
VarPointsTo

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 datalog specification → datalog engine

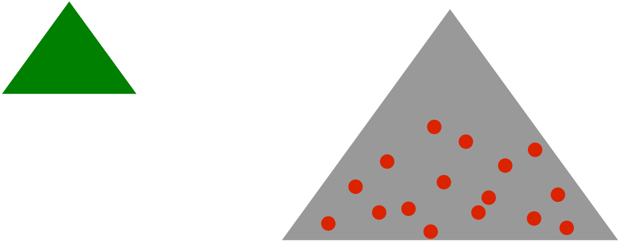
inefficient?

traditional design of single algorithm major human effort!

always use index efficiently

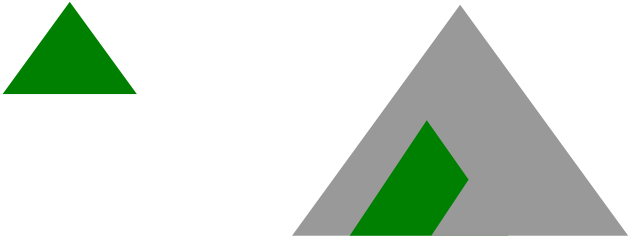
always use index efficiently

- Δ`VarPointsTo(?from, ?obj)` $\bowtie$ `Assign(?from, ?to)`



always use index efficiently

- Δ`VarPointsTo(?from, ?obj)` $\bowtie$ `Assign(?from, ?to)`
- Δ`VarPointsTo(?from, ?obj)` $\bowtie$ `Assign(?to, ?from)`



heuristics for searching algorithm space

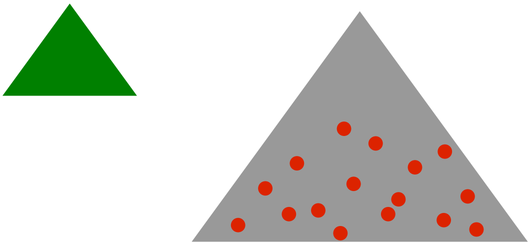
always use index efficiently

never iterate over full views

always use index efficiently

never iterate over full views

- Δ`Assign(?to, ?from)` $\bowtie$ `VarPointsTo(?from, ?obj)`

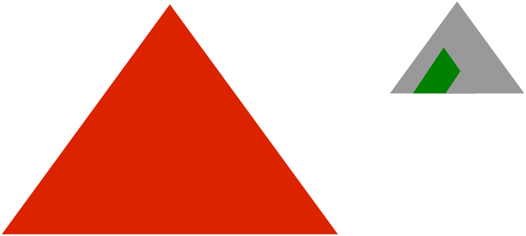


heuristics for searching algorithm space

always use index efficiently

never iterate over full views

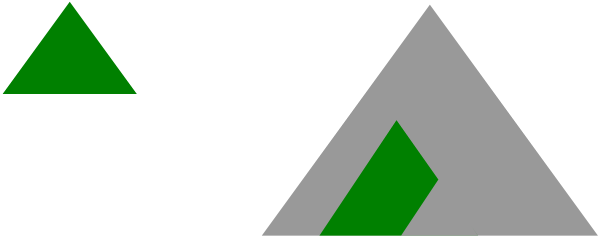
- `ΔAssign(?to, ?from)` $\bowtie$ `VarPointsTo(?from, ?obj)`
- `VarPointsTo(?from, ?obj)` $\bowtie$ `ΔAssign(?to, ?from)`



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

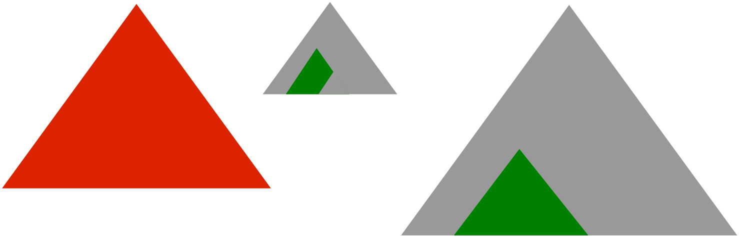
heuristics for searching algorithm space

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).
```

folding transformation: example

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).
```

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

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

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

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!

future work

pointer analysis

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

future work

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

future work

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!

extensible and domain-specific languages

embedding domain-specific languages
[OOPSLA 2004, GPCE 2005, OOPSLA 2006, OOPSLA 2008, ATEM 2007]
preventing injection attacks [GPCE 2007, SCP 2009]
parse table composition [SLE 2008]

program transformation systems

stratego/xt transformation system [PEPM 2006, FI 2006, SCP 2008]
grammar engineering [LDTA 2007]

program analysis

scalable declarative pointer analysis [ISSTA 2009]