Declarative Static Analysis with Doop

Yannis Smaragdakis
George Kastrinis

also work of
Martin Bravenboer
George Balatsouras

European Research Council
Established by the European Commission

European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS, CULTURE & SPORTS
MANAGING AUTHORITY
Co-financed by Greece and the European Union
NSRF 2007-2013
programme for development
EUROPEAN SOCIAL FUND

LOGICBLOX

Overview

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

Pointer Analysis

- What objects can a variable point to?

program

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

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

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

points-to

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

objects represented by allocation sites

Pointer Analysis

- What objects can a variable point to?

program

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

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

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

points-to

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

Pointer Analysis

- What objects can a variable point to?

program

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

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

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

points-to

foo:a	new A1()
bar:a	new A2()
id:a	n
foo:b	n
bar:b	n

remember for later: context-sensitivity is what makes an analysis precise

context-sensitive points-to

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

Pointer Analysis: A Complex Domain

flow-sensitive

field-sensitive

heap cloning

context-sensitive

binary decision diagrams

inclusion-based

unification-based

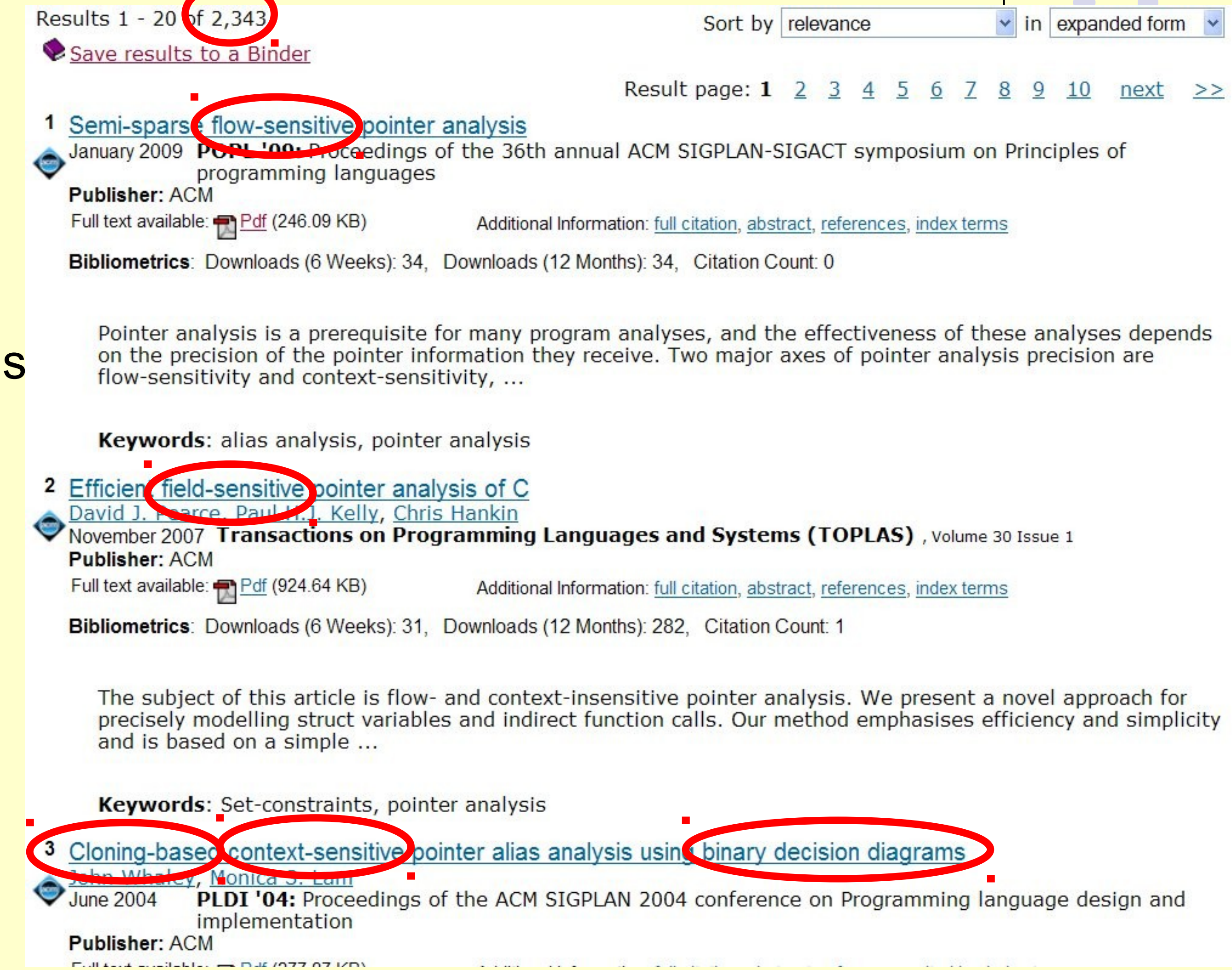
on-the-fly call graph

k-cfa

object sensitive

field-based

demand-driven



Results 1 - 20 of 2,343 Sort by relevance in expanded form

Save results to a Binder

Result page: **1** 2 3 4 5 6 7 8 9 10 next >>

1 Semi-sparse flow-sensitive pointer analysis
January 2009 **POPL '09:** Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages
Publisher: ACM
Full text available: Pdf (246.09 KB) Additional Information: full citation, abstract, references, index terms
Bibliometrics: Downloads (6 Weeks): 34, Downloads (12 Months): 34, Citation Count: 0

Pointer analysis is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the pointer information they receive. Two major axes of pointer analysis precision are flow-sensitivity and context-sensitivity, ...

Keywords: alias analysis, pointer analysis

2 Efficient field-sensitive pointer analysis of C
David J. Pearce, Paul H. J. Kelly, Chris Hankin
November 2007 **Transactions on Programming Languages and Systems (TOPLAS)** , Volume 30 Issue 1
Publisher: ACM
Full text available: Pdf (924.64 KB) Additional Information: full citation, abstract, references, index terms
Bibliometrics: Downloads (6 Weeks): 31, Downloads (12 Months): 282, Citation Count: 1

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

Keywords: Set-constraints, pointer analysis

3 Cloning-based context-sensitive pointer alias analysis using binary decision diagrams
John Whaley, Monica S. Lam
June 2004 **PLDI '04:** Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation
Publisher: ACM

Our Framework

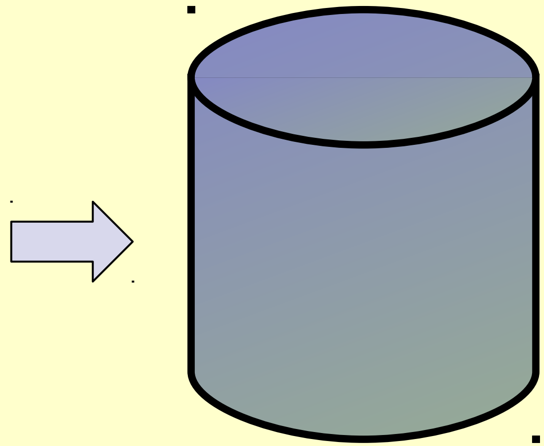
- Datalog-based pointer analysis framework for Java
- Declarative: what, not how
- Sophisticated, very rich set of analyses
  - subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive heap, abstraction, type filtering, precise exception analysis
- Support for full semantic complexity of Java
  - jvm initialization, reflection analysis, threads, reference queues, native methods, class initialization, finalization, cast checking, assignment compatibility

http://doop.program-analysis.org

Datalog: Declarative Mutual Recursion

source

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



Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

rules

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

head

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

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

Datalog: Declarative Mutual Recursion

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

head relation

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

bodies

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

body relations

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

join variables

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

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

Datalog: Declarative Mutual Recursion

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

recursion

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

VarPointsTo

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

Move

a	b
b	a
c	b

1st rule result

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

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

2nd rule evaluation

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

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

2nd rule result

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

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

Datalog: Declarative Mutual Recursion

source

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

Alloc

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

Move

a	b
b	a
c	b

VarPointsTo

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

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

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

Datalog: Properties

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

Less programming, more specification

Modeling Flavors of Context-Sensitivity

Expressiveness and Insights

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

A General Formulation of Context-Sensitive Analyses

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

Simple Intermediate Language

- We consider Java-bytecode-like language
  - allocation instructions (`Alloc`)
  - local assignments (`Move`)
  - virtual and static calls (`VCall`, `SCall`)
  - field access, assignments (`Load`, `Store`)
  - standard type system and symbol table info (`Type`, `Subtype`, `FormalArg`, `ActualArg`, etc.)

Rule 1: Allocating Objects (Alloc)

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

obj: var = new Something();

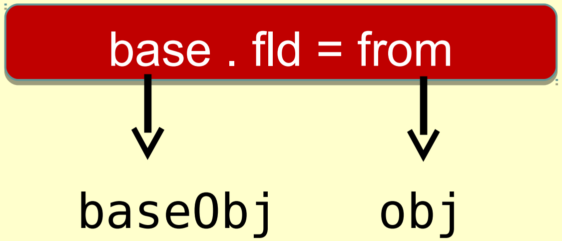
Rule 2: Variable Assignment
(Move)

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

to = from

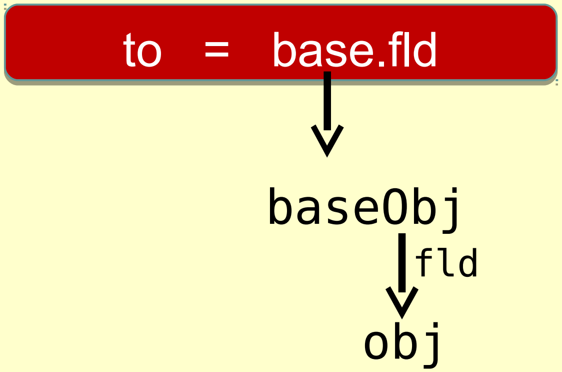
Rule 3: Object Field Write
(Store)

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



Rule 4: Object Field Read (Load)

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



Rule 5: Static Method Calls (`SCall`)

```
MergeStatic(invo, callerCtx) = calleeCtx,
Reachable(toMeth, calleeCtx),
CallGraph(invo, callerCtx, toMeth, calleeCtx)
<-
  SCall(toMeth, invo, inMeth),
  Reachable(inMeth, callerCtx).
```

invo: toMeth(..)

Rule 6: Virtual Method Calls (VCall)

```
Merge(obj, hctx, invo, callerCtx) = calleeCtx,
Reachable(toMeth, calleeCtx),
VarPointsTo(this, calleeCtx, obj, hctx),
CallGraph(invo, callerCtx, toMeth, calleeCtx)
<-
  VCall(base, sig, invo, inMeth),
  Reachable(inMeth, callerCtx),
  VarPointsTo(base, callerCtx, obj, hctx),
  LookUp(obj, sig, toMeth),
  ThisVar(toMeth, this).
```

invo: base.sig(..)

$$\texttt{obj} \xrightarrow{\texttt{sig}} \texttt{toMeth}$$

Rule 7: Parameter Passing

```
InterProcAssign(to, calleeCtx, from, callerCtx)
<-
  CallGraph(invo, callerCtx, meth, calleeCtx),
  ActualArg(invo, i, from),
  FormalArg(meth, i,to).
```

invo: meth(.., from, ..) → meth(.., to, ..)

Rule 8: Return Value Passing

```
InterProcAssign(to, callerCtx, from, calleeCtx)
<-
  CallGraph(invo, callerCtx, meth, calleeCtx),
  ActualReturn(invo, to),
  FormalReturn(meth, from).
```

invo: to = meth(..) → meth(..) { .. return from; }

Rule 9: Parameter/Result Passing as Assignment

```
VarPointsTo(to, toCtx, obj, hctx)
<-
  InterProcAssign(to, toCtx, from, fromCtx),
  VarPointsTo(to, toCtx, obj, hctx).
```

Can Now Express Past Analyses Nicely

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

Can Now Express Past Analyses Nicely

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

Can Now Express Past Analyses Nicely

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

Lots of Insights and New Algorithms

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

Getting Performance Out of Datalog Rules

Doop Can Achieve Great Performance

- Where is the magic? In very few places!
  - 4 orders of magnitude via optimization methodology for highly recursive Datalog!
    - straightforward data processing optimization (indexes), but with an understanding of how Datalog does recursive evaluation
  - no BDDs
    - are they needed for pointer analysis?
  - simple domain-specific enhancements that increase both precision and performance in a direct (non-BDD) implementation

Datalog: Naïve Evaluation

- Datalog

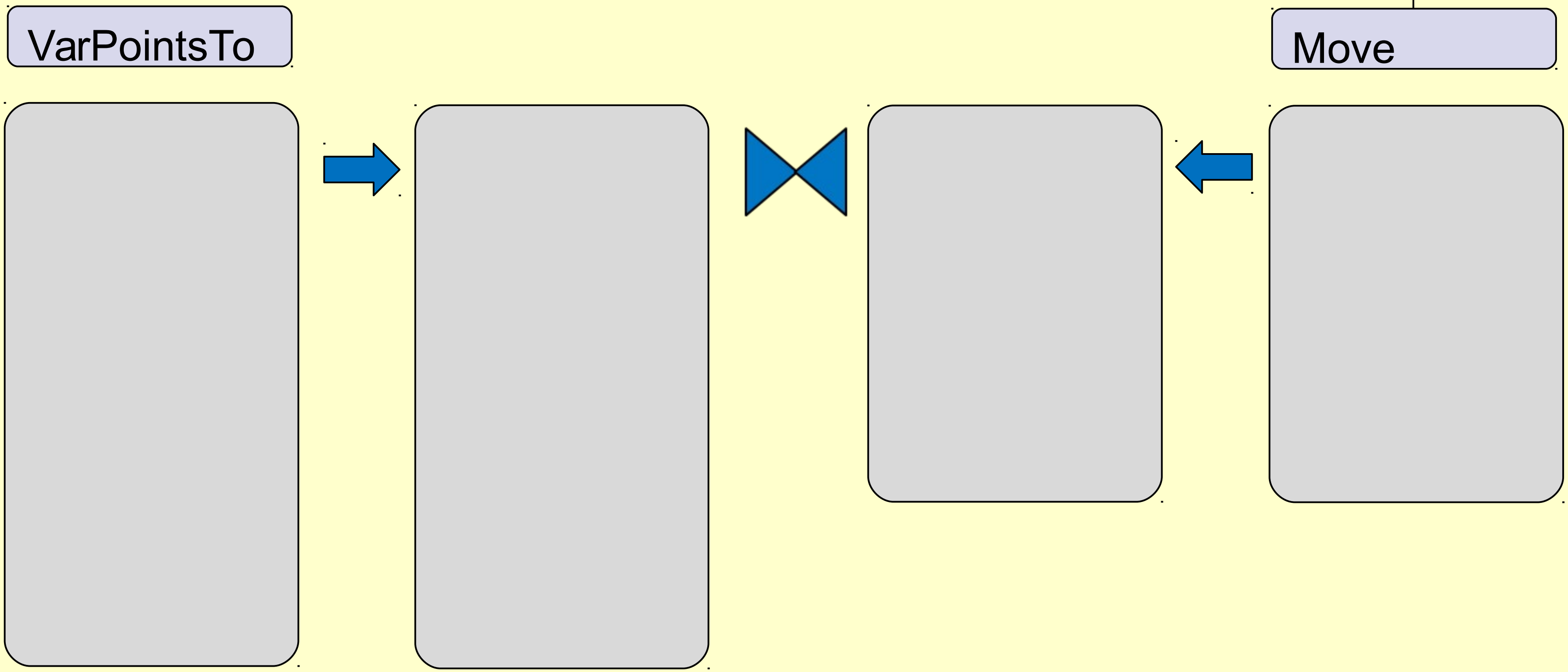
```
VarPointsTo(var, obj) <-
  Alloc(var, obj).

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

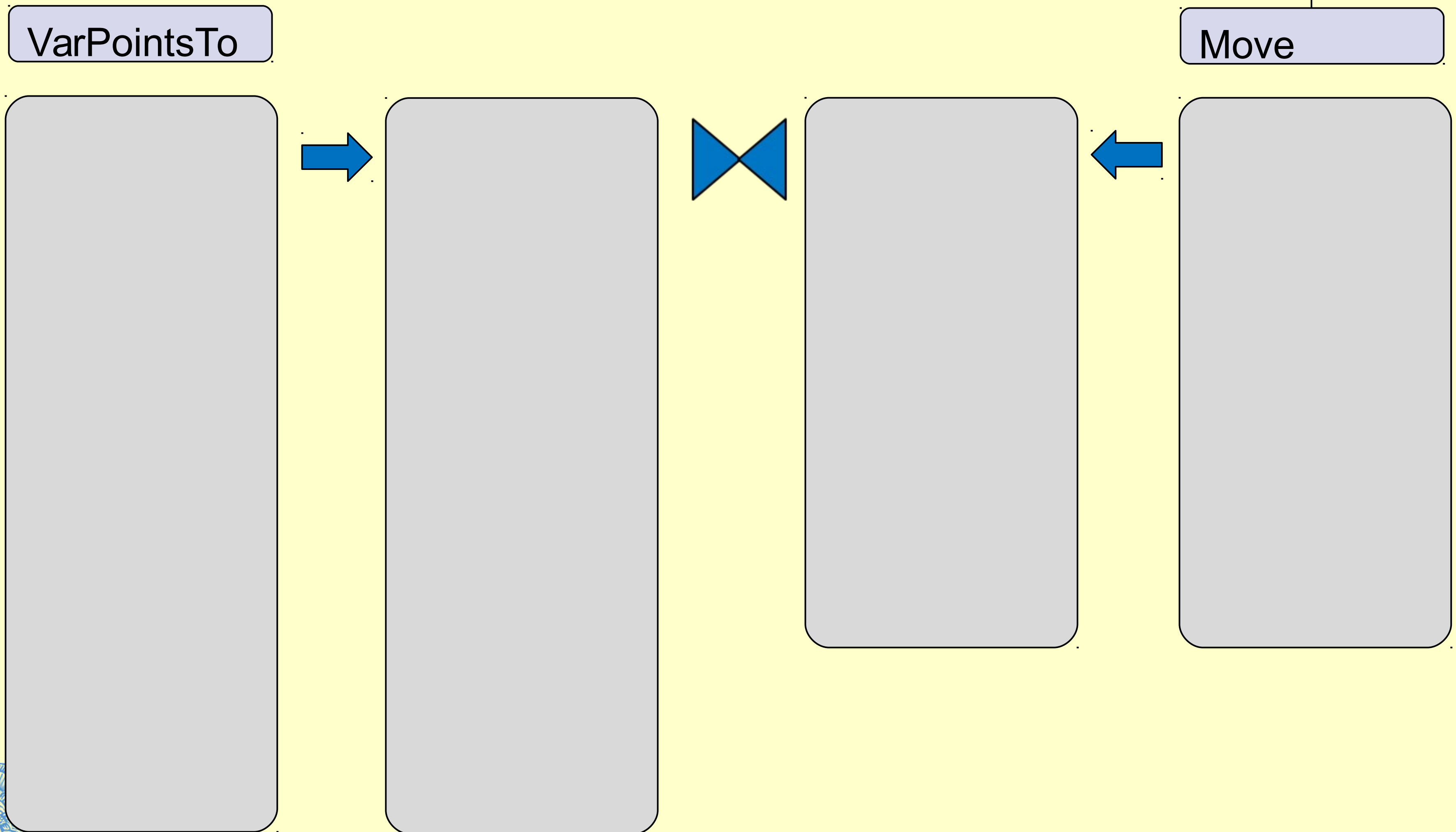
- Naïve Evaluation (relational algebra)

```
VarPointsTo := Alloc
repeat
  tmp := π_{to→var,obj}(VarPointsTo ∞ _{from=var} Move)
  VarPointsTo := VarPointsTo ∪ tmp
until fixpoint
```

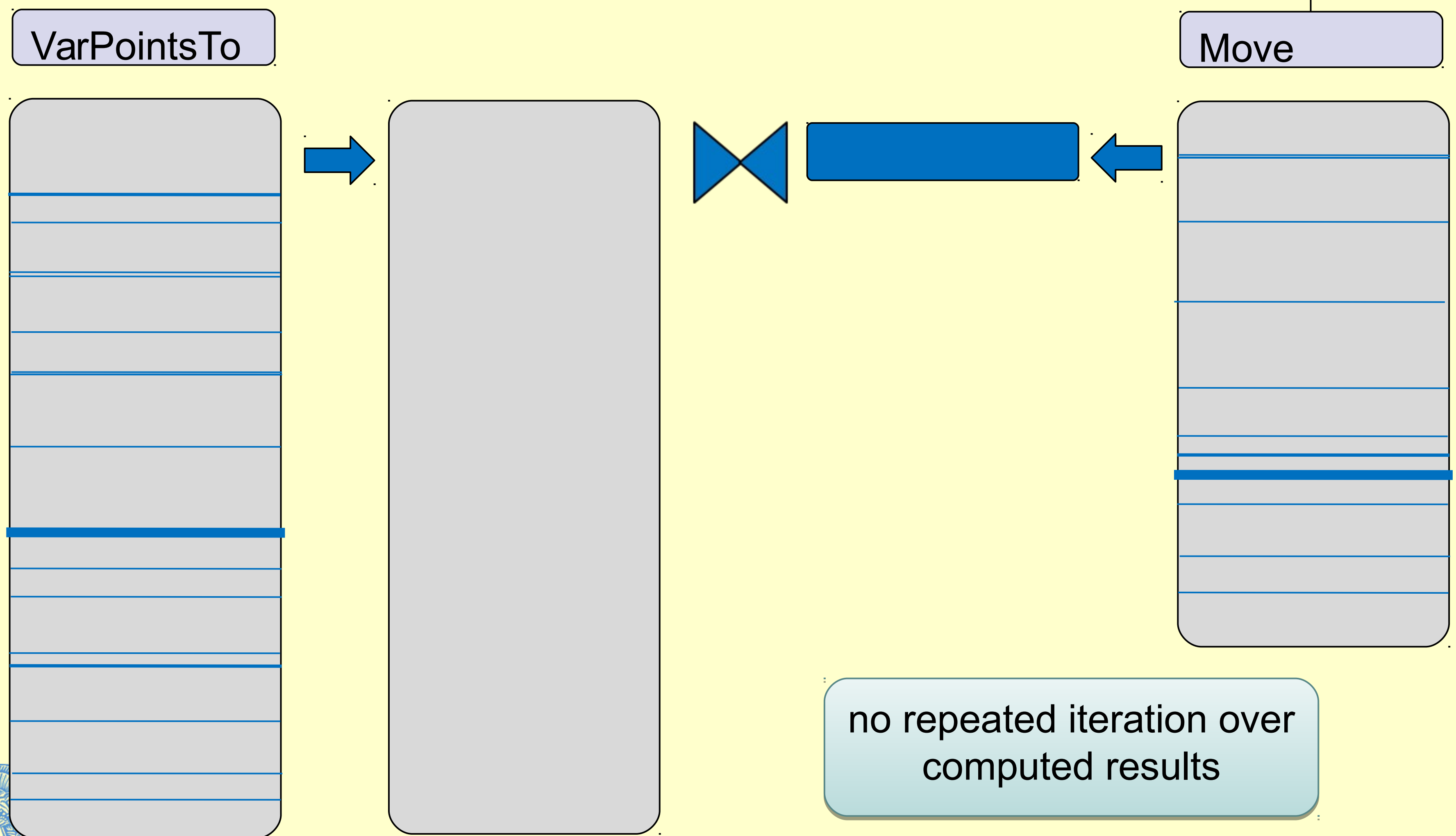
Datalog: Naïve Evaluation



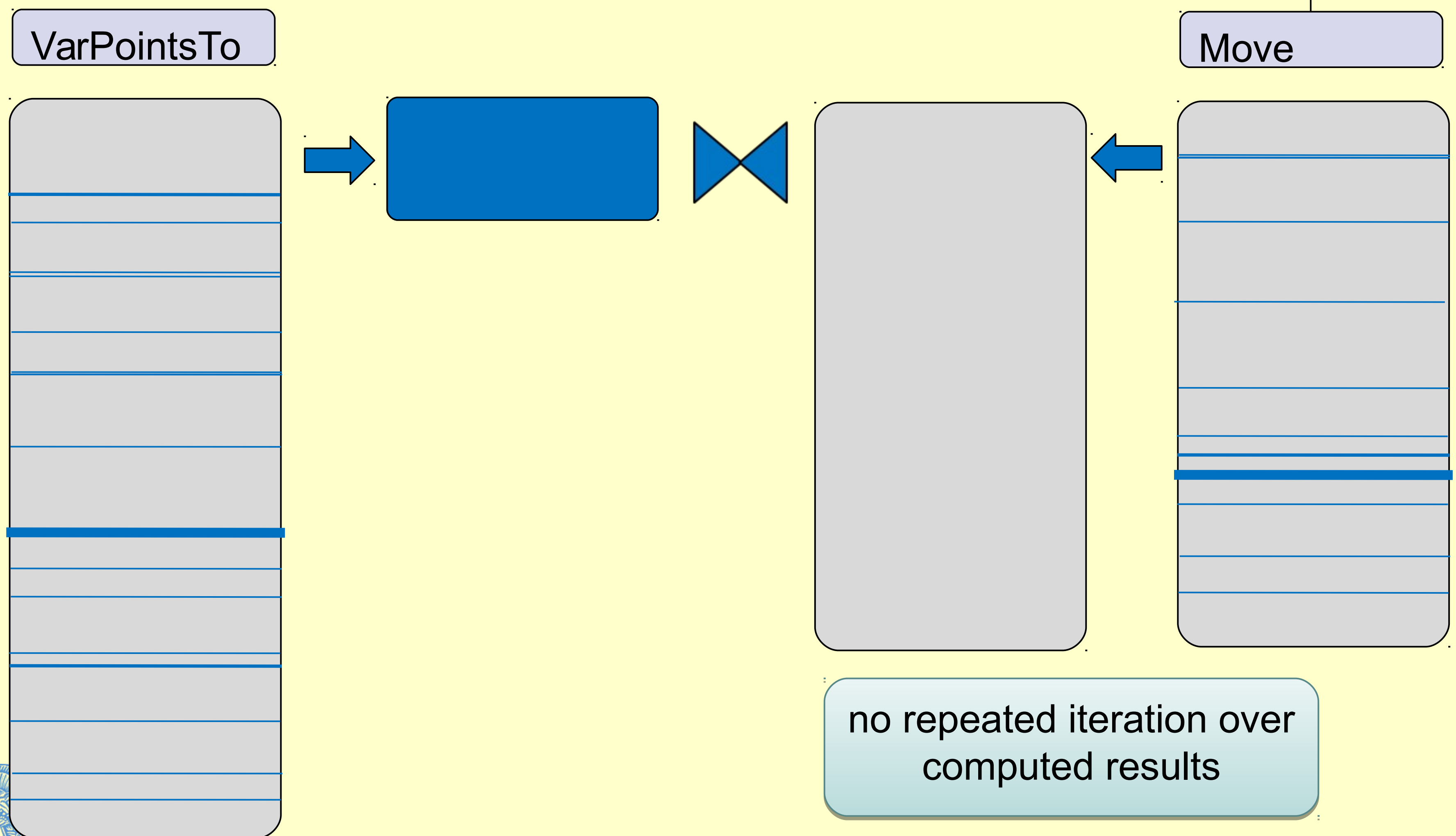
Datalog: Naïve Evaluation



Datalog: Semi-Naïve Evaluation



Datalog: Semi-Naïve Evaluation



no repeated iteration over computed results

Optimization Idea: Optimize Indexing for Semi- Naïve Evaluation

- Datalog rule

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

- Semi-Naïve Evaluation

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

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

Optimization Insight

- For highly recursive Datalog programs, ***relation deltas produced by semi-naive evaluation should bind all the variables needed to index into other relations***
- Can require complex reasoning
  - whole program optimization
  - human needs to be in the loop in the search of algorithm space!

Optimizations = Datalog Source Transformations

- The Datalog engine we use exposes indexing
  - argument order determines index
- Essentially, the engine does not represent relations, only indexes (as b-trees)
  - i.e., functions
  - very much like a vertical/column store in DBs, but with everything exposed to the Datalog language level

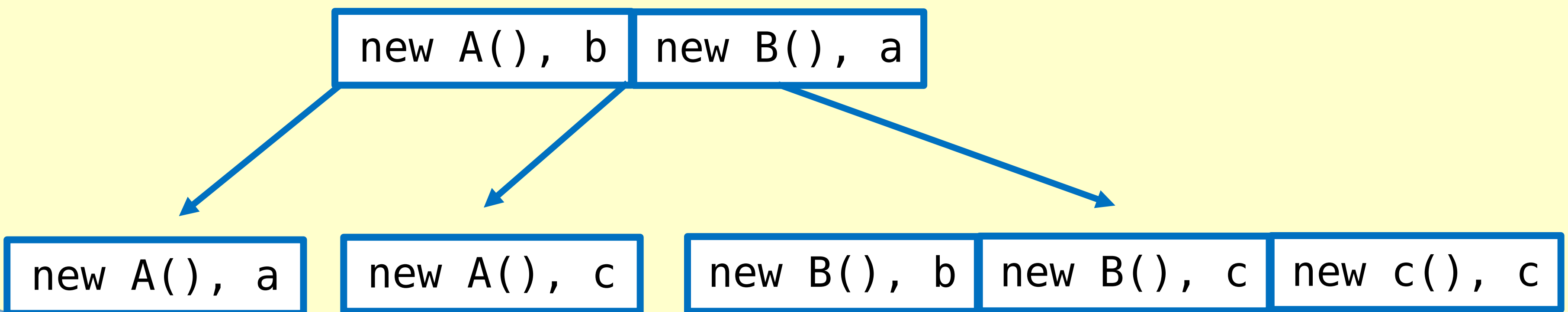
LOGICBLOX

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()

index organized by reverse argument order

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

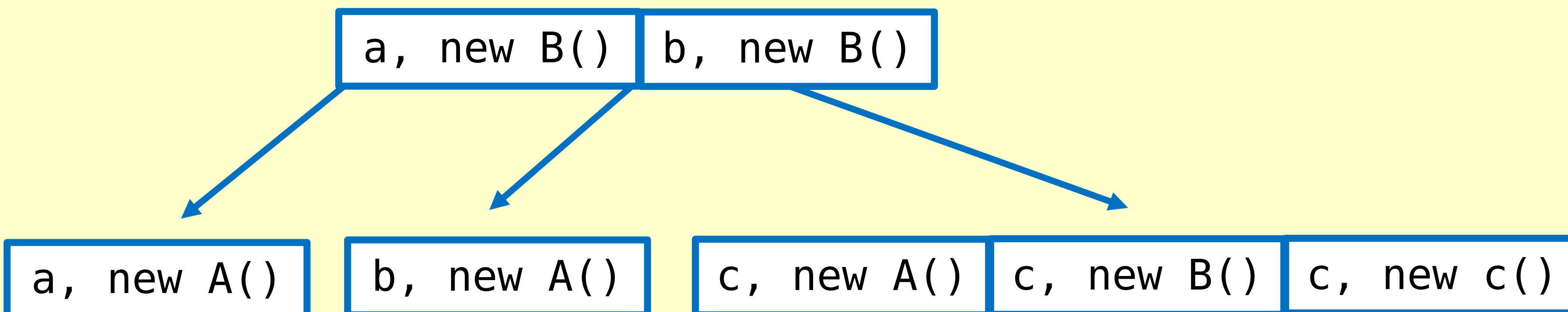


Relations As Multidimensional Indexes

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

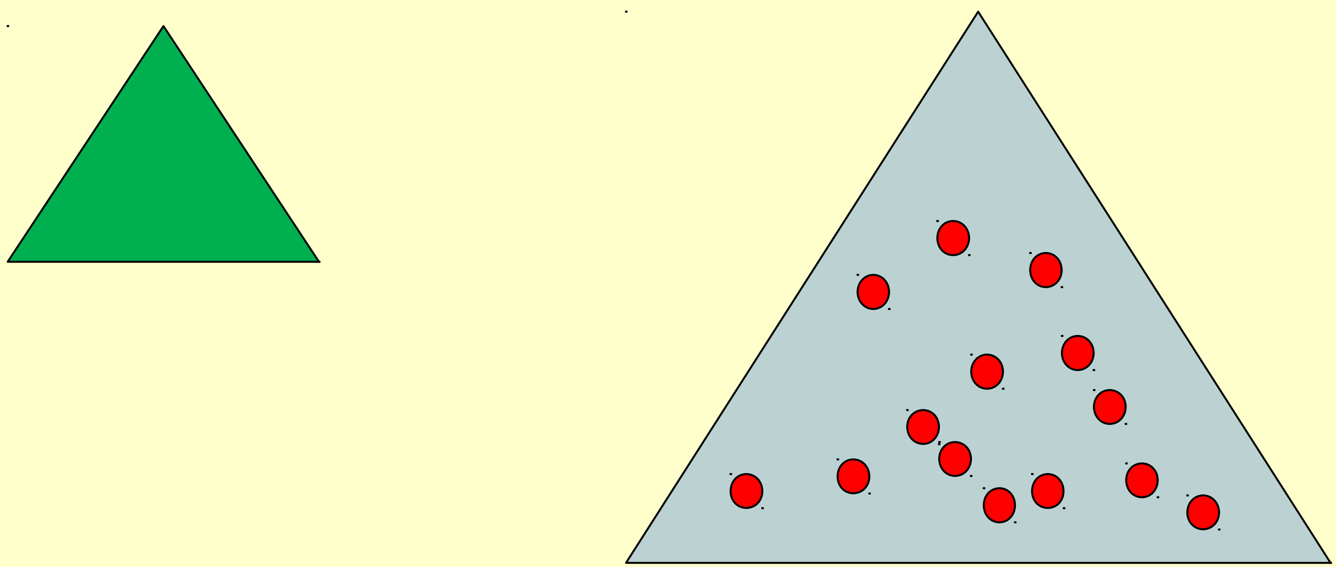
```
VarPointsTo(obj, var)
```



Heuristics For Searching Algorithm Space

- Always use index efficiently

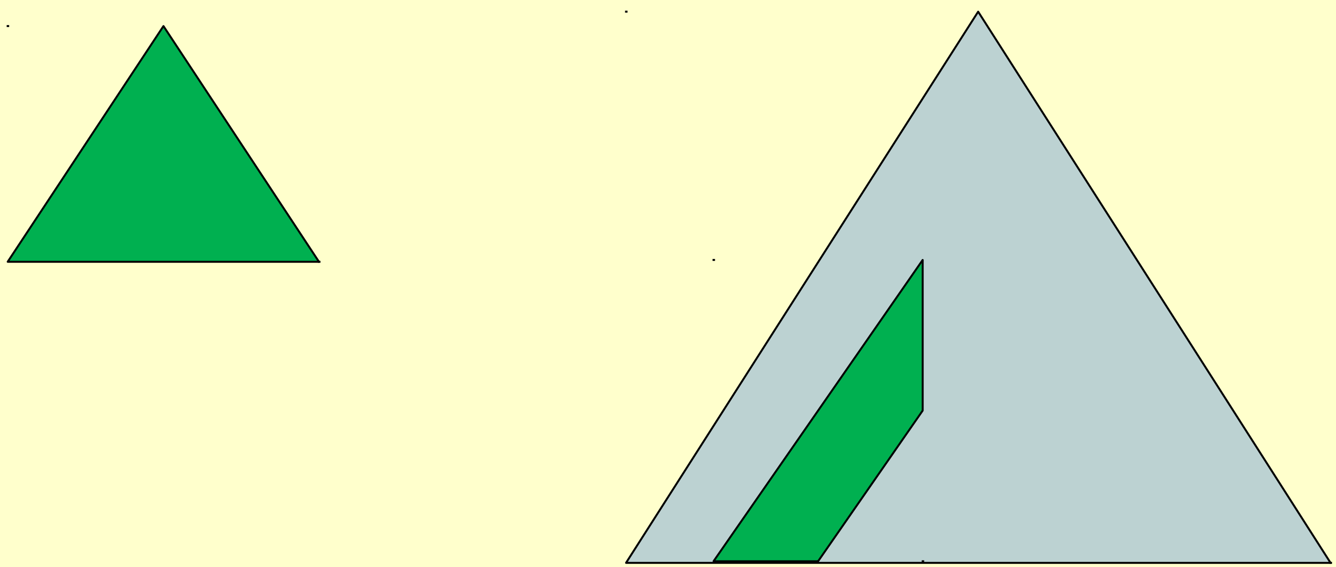
  `ΔVarPointsTo(from, obj) ⋈ Move(from, to)`



Heuristics For Searching Algorithm Space

- Always use index efficiently

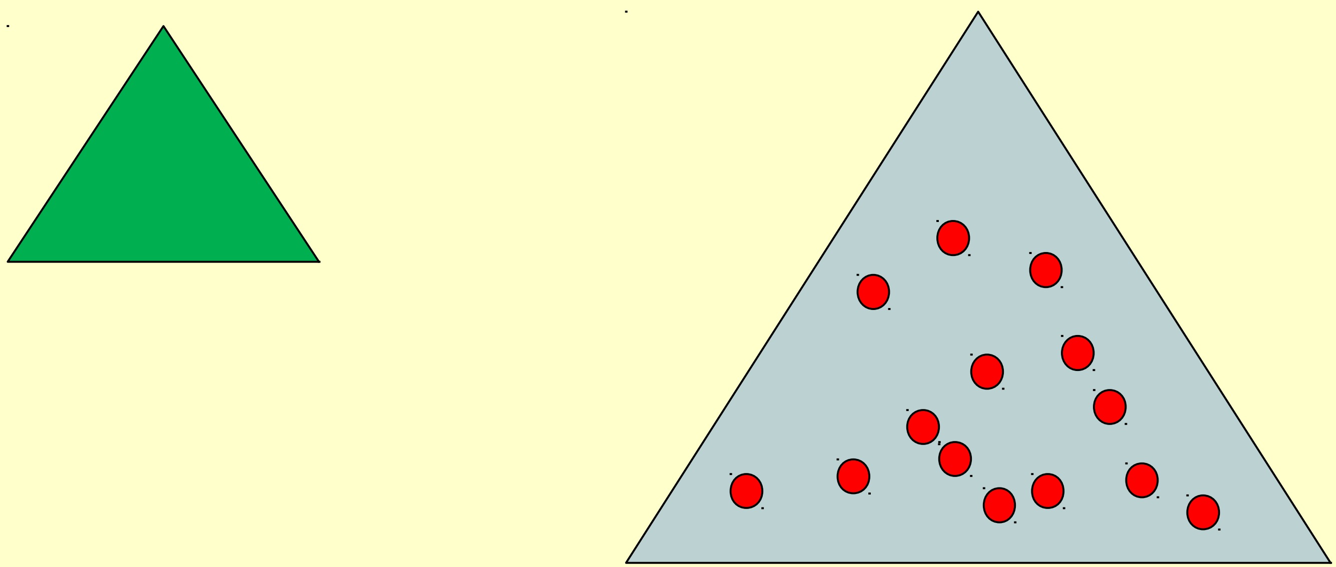
`ΔVarPointsTo(from, obj) ∞ Move(to, from)`



Heuristics For Searching Algorithm Space

- Never iterate over full views

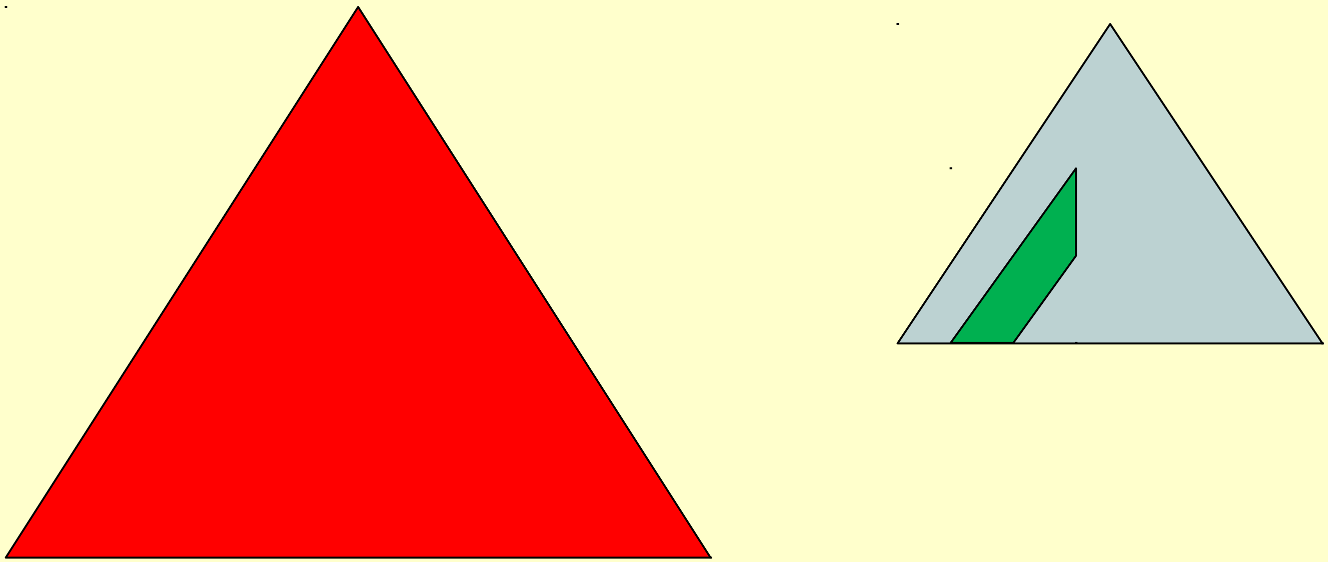
`ΔMove(to, from) ∞ VarPointsTo(from, obj)`



Heuristics For Searching Algorithm Space

- Never iterate over full views

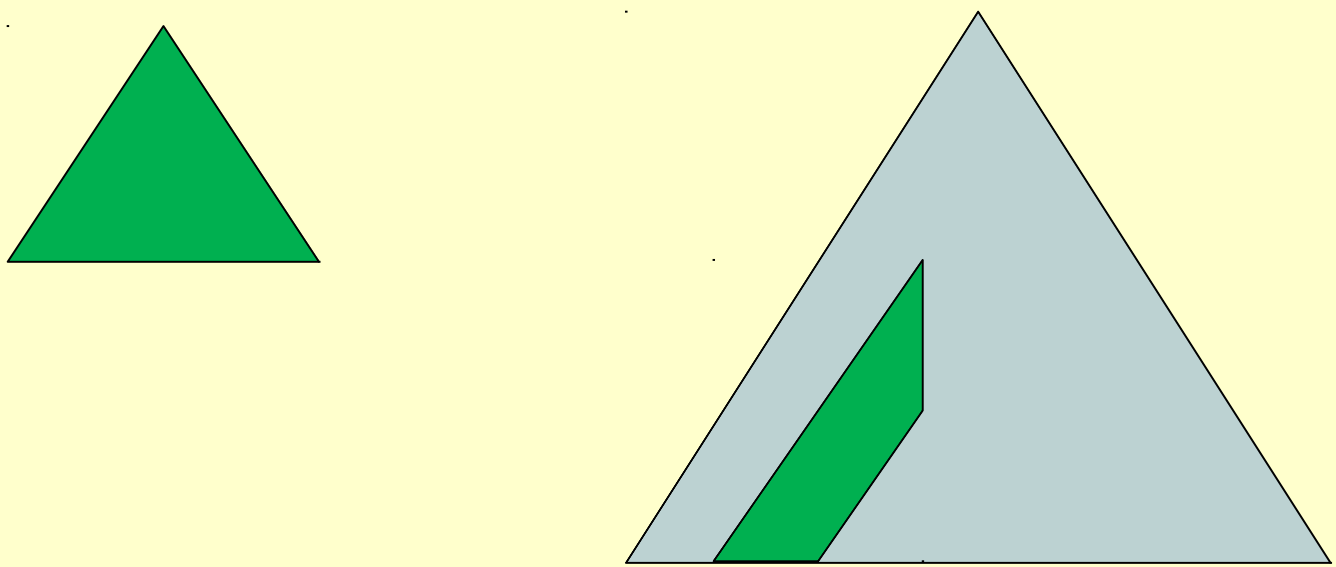
```
ΔMove(to, from) ∞ VarPointsTo(from, obj)
```



Heuristics For Searching Algorithm Space

- Never iterate over full views

```
ΔVarPointsTo(obj, from) ∞ Move(to, from)
  ΔMove(to, from) ∞ VarPointsTo(obj, from)
```



Not Always Easy

- Specification

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

- Semi-naïve executions

```
ΔFldPointsTo(baseObj, fld, obj) <-
  Store(base, fld, from),
  ΔVarPointsTo(baseObj, base),
  VarPointsTo(obj, from).

ΔFldPointsTo(baseObj, fld, obj) <-
  Store(base, fld, from),
  VarPointsTo(baseObj, base),
  ΔVarPointsTo(obj, from).
```

Not Always Easy: Whole Program Property

- Specification

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

- Join orderings

```
ΔvarPointsTo(baseObj, base)
∞ Store(base, fld, from)
∞  VarPointsTo(obj, from)

ΔvarPointsTo(obj, from)
∞ Store(base, fld, from)
∞ VarPointsTo(baseObj, base)
```

there is no efficient variable ordering for `StoreField` (`?base` and `?from` cannot both be last)

Solution: Introduce New Index/Relation

- Specification

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

- Optimized rules

```
FieldPointsTo(baseObj, fld, from) <-
  StoreObjectField(baseObj, fld, from),
  VarPointsTo(obj, from).

StoreObjectField(baseObj, fld, from) <-
  Store(from, fld, base),
  VarPointsTo(baseObj, base).
```

Set-Based Pre-Analysis
a universal optimization technique for flow-insensitive analyses

Set-Based Pre-Analysis

Idea: can do much reasoning at the *set level* instead of the *value level*

can simplify the program as a result

a local transformation

think of it as creating a normal form (or IR) for points-to analysis

“hello, world” Example

$P \supseteq Q$

$\wedge$

$R \supseteq P$

$\Downarrow$

$R \supseteq Q$

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

(MOVE elimination)

Simple subset reasoning

statement redundant for *analysis* purposes

"hello, world" Example

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

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

(MOVE elimination)

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

More Examples

```
r = q;
p.f = r;
p.f = q;  // redundant
```
(STORE elimination)

```
p.f = q;
r = p.f
r = q;    // redundant
```
(another MOVE elimination)

Even More Examples

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

(LOAD elimination)

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

(CALL elimination!!!)

Not Even Close To Done

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

(ARRAYLOAD elimination—
for array insensitive analyses)

can apply all previous patterns in combination with array ops, or with static loads, calls, stores, etc.

transforms apply to fixpoint (one enables others)

And Also…

Duplicate variable elimination

same as past work using the constraint graph to merge points-to sets

Constraint Graph:

∀ p = q; $Q \longrightarrow P$

E.g.,

- merge vars in same strongly connected component of constraint graph [Faehndrich et al.]
- merge vars with identical in-flows [Rountev and Chandra, Hardekopf and Lin]
- merge vars with same dominator [Nasre]

Transform Effect, Pictorially

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

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

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

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

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

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

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

          if @param0 == null goto label4;

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

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

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

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

Observations

The reduced program is NOT valid for execution

only for flow-insensitive points-to analysis

Set-based reasoning makes sense since points-to analyses are expressible via subset constraints

MOVE elimination follows from MOVE rule in analysis

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

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

So, How Well Does This Work?

Over many analyses, DaCapo benchmarks

(ctx-insens, 1call, 1call+H, 1obj, 1obj+H, 2obj+H, 2type+H)

20% average speedup

(median: 20%, max: 110%)

Eliminates ~30% of local vars

Decimates (97% elimination!) MOVE instructions

Eliminates more than 30% of context-sensitive points-to facts