Declarative Static Program Analysis with Doop

Jonathan Aldrich
17-355/17-665/17-819: Program Analysis

Slides adapted by permission from Yannis Smaragdakis
Based on work with Martin Bravenboer, George Kastrinis, George Balatsouras
Overview

● Declarative pointer analysis
  ● Specified as logical rules in Datalog
  ● Supports various forms of context-sensitivity

● Efficient implementation
  ● Datalog execution model
  ● Optimization of Datalog queries

● Why do we care?
  ● Easy to understand, modify, and optimize
  ● Best analysis performance available today
## Pointer Analysis

- What objects can a variable point to?

<table>
<thead>
<tr>
<th>Program</th>
<th>Points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo() {</td>
<td>foo:a new A1()</td>
</tr>
<tr>
<td>Object a = new A1();</td>
<td>bar:a new A2()</td>
</tr>
<tr>
<td>Object b = id(a);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>void bar() {</td>
<td></td>
</tr>
<tr>
<td>Object a = new A2();</td>
<td></td>
</tr>
<tr>
<td>Object b = id(a);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Object id(Object a) {</td>
<td></td>
</tr>
<tr>
<td>return a;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Pointer Analysis

- What objects can a variable point to?

```
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;
}
```

<table>
<thead>
<tr>
<th>program</th>
<th>points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo() {</td>
<td>foo:a</td>
</tr>
<tr>
<td>Object a = new A1();</td>
<td>bar:a</td>
</tr>
<tr>
<td>Object b = id(a);</td>
<td>id:a</td>
</tr>
</tbody>
</table>
**Pointer Analysis**

- What objects can a variable point to?

<table>
<thead>
<tr>
<th>program</th>
</tr>
</thead>
</table>
| 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;  
} |

<table>
<thead>
<tr>
<th>points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:a</td>
</tr>
<tr>
<td>bar:a</td>
</tr>
<tr>
<td>id:a</td>
</tr>
<tr>
<td>foo:b</td>
</tr>
<tr>
<td>bar:b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>context-sensitive points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:a</td>
</tr>
<tr>
<td>bar:a</td>
</tr>
<tr>
<td>id:a (foo)</td>
</tr>
<tr>
<td>id:a (bar)</td>
</tr>
<tr>
<td>foo:b</td>
</tr>
<tr>
<td>bar:b</td>
</tr>
</tbody>
</table>

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

Yannis Smaragdakis  
University of Athens
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.

Yannis Smaragdakis
University of Athens
Algorithms Found in a 10-Page Pointer Analysis Paper

Yannis Smaragdakis
University of Athens

variation points unclear

every variant a new algorithm

correctness unclear

incomparable in precision

/* Alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 2 */

procedure false_for_deleting_assign(N)
N: a pointer assignment to be deleted;
begin
1. N: a pointer assignment to be added;
M: the statement after which statement N is added;
2. begin
3. 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);

end
6. for each may_hold(M, AA, PA = (o1, o2)) = YES, and may_hold(N, AA, PA) = NO
add (M, AA, PA) to worklist;

begin
7. reiterate_worklist(worklist, FALSIFIED);

end

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

Figure 1: Excerpt from page of the paper with three boxes highlighting three key points: variation points unclear, every variant a new algorithm, and correctness unclear.

Figure 5: Reiteration for the incremental algorithm

Figure 4: Reintroduce aliases for naive falsification

Figure 7: Procedures for falsifying aliases which are not
Program Analysis: a Domain of Mutual Recursion

\[ x = y \]

\[ \text{var points-to} \]
Program Analysis: a Domain of Mutual Recursion

\[ x = f() \]

var points-to

call graph

Yannis Smaragdakis
University of Athens
Program Analysis: a Domain of Mutual Recursion

\[
x = y.f()
\]

var points-to

call graph

Yannis Smaragdakis
University of Athens
Program Analysis: a Domain of Mutual Recursion

\[
x = \text{new } A()
\]

- **var points-to**
- **call graph**
- **reachable methods**
Program Analysis: a Domain of Mutual Recursion

\[ x.f = y \]

- call graph
- var points-to
- field points-to
- reachable methods
Program Analysis: a Domain of Mutual Recursion

\[ x = y.f \]

Diagram:
- x = y.f
- var points-to
- call graph
- field points-to
- reachable methods
Program Analysis: a Domain of Mutual Recursion

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

throw e
Program Analysis: a Domain of Mutual Recursion

catch(E e)

var points-to

call graph

reachable methods

exceptions

field points-to

var points-to

Yannis Smaragdakis
University of Athens
Program Analysis: a Domain of Mutual Recursion

var points-to

call graph  exceptions

field points-to

reachable methods

g()
Program Analysis: a Domain of Mutual Recursion

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

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

Alloc

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

Move

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

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

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

Yannis Smaragdakis
University of Athens
## Datalog: Declarative Mutual Recursion

### Source

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

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

VarPointsTo(to, obj) <=
    Move(to, from),
    VarPointsTo(from, obj).
```
Datalog: Declarative Mutual Recursion

\[\begin{align*}
\text{source} & : \\
a & = \text{new } A(); \\
b & = \text{new } B(); \\
c & = \text{new } C(); \\
a & = b; \\
b & = a; \\
c & = b; \\
\text{Alloc} & : \\
a & \mid \text{new } A() \\
b & \mid \text{new } B() \\
c & \mid \text{new } C() \\
\text{VarPointsTo} & : \\
(\text{VarPointsTo}(\text{var, obj}) \leftarrow \text{Alloc}(\text{var, obj}). \\
(\text{VarPointsTo}(\text{to, obj}) \leftarrow \\
\text{Move}(\text{to, from}), \\
\text{VarPointsTo}(\text{from, obj}).
\end{align*}\]
# 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

```
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

Move
a | b
b | a
c | b

\[
\text{VarPointsTo}(\text{var}, \text{obj}) \leftarrow \text{Alloc}(\text{var}, \text{obj}).
\]

\[
\text{VarPointsTo}(\text{to}, \text{obj}) \leftarrow \\
\text{Move}(\text{to}, \text{from}), \\
\text{VarPointsTo}(\text{from}, \text{obj}).
\]
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

Move
a | b
b | a
c | b

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

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

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

Yannis Smaragdakis
University of Athens
Datalog: Declarative Mutual Recursion

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

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

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

VarPointsTo

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

Move

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

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

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

2nd rule evaluation
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()
a | new B()

Move
a | b
b | a
c | b

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

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

2nd rule result
## Datalog: Declarative Mutual Recursion

<table>
<thead>
<tr>
<th>Source</th>
<th>Alloc</th>
<th>VarPointsTo</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = new A();</code></td>
<td>`a</td>
<td>new A()`</td>
</tr>
<tr>
<td><code>b = new B();</code></td>
<td>`b</td>
<td>new B()`</td>
</tr>
<tr>
<td><code>c = new C();</code></td>
<td>`c</td>
<td>new C()`</td>
</tr>
<tr>
<td><code>a = b;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>b = a;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>c = b;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **VarPointsTo(var, obj)** <- Alloc(var, obj).
- **VarPointsTo(to, obj)** <- Move(to, from), VarPointsTo(from, obj).

Yannis Smaragdakis  
University of Athens
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

Yannis Smaragdakis
University of Athens
The **DOOP** 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
Some Doop Contributions

- Expressed complete, complex pointer analyses in Datalog
  - core specification: ~1500 logic rules
  - parameterized by a handful of rules per analysis flavor
- Synthesized efficient algorithms from specification
  - order of magnitude performance improvement
  - allowed to explore more analyses than past literature
- Approach: heuristics for searching algorithm space
  - targeted at recursive problem domains
- Demonstrated scalability with explicit representation
  - Contrast: previous work used BDDs (bddbd, Paddle)
Not Expected

- Expressed complete, complex pointer analyses in Datalog
  
  “[E]ncoding all the details of a complicated program analysis problem [on-the-fly call graph construction, handling of Java features] purely in terms of subset constraints may be difficult or impossible.” (Lhotak)

- Scalability and Efficiency
  
  “Efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in data structures and algorithms”
How Datalog Executes*

*Note: we will focus on bottom-up execution, as used in Doop. A top-down execution strategy is possible, and is useful for other applications of Datalog
Naïve Datalog Execution

- Execute each rule against all facts in the current database
- Add the resulting facts to the database
- Repeat until no more facts are added

- Efficiency problem
  - Facts inferred in the first iteration are re-inferred in all subsequent iterations
  - In second iteration, need only consider rule instantiations based on at least one new fact
Semi-Naïve Datalog Execution

- Execute program incrementally
  - For each rule r, for each fact f produced in relation R in the previous round, generate all facts inferable via rule r, fact f, and all current facts for relations other than R

- Example
  - Original program
    \[
    \varpoints{(to, obj)} <- \\
    \text{Move}(to, from), \\
    \varpoints{(from, obj)}.
    \]
  - Semi-Naïve rewriting
    \[
    \Delta \varpoints{(to, obj)} <- \\
    \text{Move}(to, from), \\
    \Delta \varpoints{(from, obj)}.
    \]
  - Note that we don't have a similar \( \Delta \) rule for Move, because Move represents program text and we never get more facts there
Semi-Naïve Datalog Execution

- Rewritten program

\[
\text{VarPointsTo}(\text{var}, \text{obj}) \leftarrow \\
\quad \text{AssignHeapAllocation}(\text{var}, \text{obj}). \]
\[
\Delta \text{VarPointsTo}(\text{to}, \text{obj}) \leftarrow \\
\quad \text{Move}(\text{to}, \text{from}), \\
\quad \Delta \text{VarPointsTo}(\text{from}, \text{obj}).
\]

- Initial facts

\[
\text{AssignHeapAllocation}("y", "o1"). \\
\text{AssignHeapAllocation}("z", "o2").
\]
\[
\text{Move}("x", "y"). \\
\text{Move}("y", "z").
\]
Practice: Semi-Naïve Execution

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

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

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

AssignHeapAllocation("x", "o1").
AssignHeapAllocation("y", "o2").

LoadField("x", "f", "x").  % x.f = x
StoreField("x", "f", "y").  % y = x.f
```
Practice: Writing Reachability Analysis

- Reachable(meth) % method implementation meth is reachable
- VarPointsTo(var, obj) % var may point to an object allocated at obj
- Alloc(var, obj, meth) % obj: var = new ...(...) in meth
- VCall(base, sig, inMeth)% virtual call: base.sig(...) in "inMeth"
- Lookup(obj, sig, meth) % looks up the method implementation for allocation % site obj and method signature sig

- var, base - variable
- obj - object allocation site
- meth, inMeth - method implementation
- sig - method name and signature

- Give rules for Reachable and VarPointsTo, assuming Alloc, VCall, and Lookup are given
Encoding Context-Sensitive Pointer Analysis in Datalog

[PLDI’10, POPL’11, CC’13, PLDI’13, PLDI’14]
Recall: Context-Sensitivity (call-site sensitivity)

- What objects can a variable point to?

<table>
<thead>
<tr>
<th>program</th>
<th>points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo() {</td>
<td>foo:a new A1()</td>
</tr>
<tr>
<td>Object a = new A1();</td>
<td>bar:a new A2()</td>
</tr>
<tr>
<td>Object b = id(a);</td>
<td>id:a new A1(), new A2()</td>
</tr>
<tr>
<td>}</td>
<td>foo:b new A1(), new A2()</td>
</tr>
<tr>
<td>void bar() {</td>
<td>bar:b new A1(), new A2()</td>
</tr>
<tr>
<td>Object a = new A2();</td>
<td></td>
</tr>
<tr>
<td>Object b = id(a);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Object id(Object a) {</td>
<td></td>
</tr>
<tr>
<td>return a;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>call-site-sensitive points-to</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:a</td>
<td>new A1()</td>
</tr>
<tr>
<td>bar:a</td>
<td>new A2()</td>
</tr>
<tr>
<td>id:a (foo)</td>
<td>new A1()</td>
</tr>
<tr>
<td>id:a (bar)</td>
<td>new A2()</td>
</tr>
<tr>
<td>foo:b</td>
<td>new A1()</td>
</tr>
<tr>
<td>bar:b</td>
<td>new A2()</td>
</tr>
</tbody>
</table>

Yannis Smaragdakis
University of Athens
Object-Sensitivity

program

class S {
    Object id(Object a) { return a; }
    Object id2(Object a) { return id(a); }
}
class C extends S {
    void fun1() {
        Object a1 = new A1();
        Object b1 = id2(a1);
    }
}
class D extends S {
    void fun2() {
        Object a2 = new A2();
        Object b2 = id2(a2);
    }
}


1-call-site-sensitive points-to

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fun1:a1</td>
<td>new A1()</td>
</tr>
<tr>
<td>fun2:a2</td>
<td>new A2()</td>
</tr>
<tr>
<td>id2:a (fun1)</td>
<td>new A1()</td>
</tr>
<tr>
<td>id2:a (fun2)</td>
<td>new A2()</td>
</tr>
<tr>
<td>id:a (id2)</td>
<td>new A1(), new A2()</td>
</tr>
<tr>
<td>id2:ret (*)</td>
<td>new A1(), new A2()</td>
</tr>
<tr>
<td>fun1:b1</td>
<td>new A1(), new A2()</td>
</tr>
<tr>
<td>fun2:b2</td>
<td>new A1(), new A2()</td>
</tr>
</tbody>
</table>

Yannis Smaragdakis
University of Athens
Object-Sensitivity

program

```java
class S {
    Object id(Object a) { return a; }
    Object id2(Object a) { return id(a); }
}
class C extends S {
    void fun1() {
        Object a1 = new A1();
        Object b1 = id2(a1);
    }
}
class D extends S {
    void fun2() {
        Object a2 = new A2();
        Object b2 = id2(a2);
    }
}
```

1-object-sensitive points-to

<table>
<thead>
<tr>
<th>Method</th>
<th>Points-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun1</td>
<td>a1, new A1()</td>
</tr>
<tr>
<td>fun2</td>
<td>a2, new A2()</td>
</tr>
<tr>
<td>id2</td>
<td>a (C1), new A1()</td>
</tr>
<tr>
<td>id2</td>
<td>a (D1), new A2()</td>
</tr>
<tr>
<td>id</td>
<td>a (C1), new A1()</td>
</tr>
<tr>
<td>id</td>
<td>a (D1), new A2()</td>
</tr>
<tr>
<td>id2:ret</td>
<td>(C1), new A1()</td>
</tr>
<tr>
<td>fun1</td>
<td>b1, new A1()</td>
</tr>
<tr>
<td>fun2</td>
<td>b2, new A2()</td>
</tr>
</tbody>
</table>

Yannis Smaragdakis
University of Athens
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)

\[
\text{VarPointsTo}(\text{var}, \text{ctx}, \text{obj}, \text{hctx}) \\
<- \\
\text{Alloc}(\text{var}, \text{obj}, \text{meth}), \\
\text{Reachable}(\text{meth}, \text{ctx}). \\
\text{where } \text{hctx} = \text{Record}(\text{obj}, \text{ctx})
\]

\textbf{obj: } \text{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)

\[\text{FldPointsTo}(\text{baseObj}, \text{baseHCtx}, \text{fld}, \text{obj}, \text{hctx}) \leftarrow \]
\[\text{Store}(\text{base}, \text{fld}, \text{from}), \]
\[\text{VarPointsTo}(\text{from}, \text{ctx}, \text{obj}, \text{hctx}), \]
\[\text{VarPointsTo}(\text{base}, \text{ctx}, \text{baseObj}, \text{baseHCtx}).\]
Rule 4: Object Field Read (Load)

\[
\text{VarPointsTo}(\text{to}, \text{ctx}, \text{obj}, \text{hctx}) \\
\leftarrow \\
\text{Load(}\text{to}, \text{base, fld)}), \\
\text{FldPointsTo(}\text{baseObj, baseHCtx, fld, obj, hctx)}), \\
\text{VarPointsTo(}\text{base, ctx, baseObj, baseHCtx}).
\]

to = base.fld

baseObj

fld

obj
Rule 5: Static Method Calls (SCall)

\[
\begin{align*}
\text{Reachable} & \ (\text{toMeth}, \ \text{calleeCtx}), \\
\text{CallGraph} & \ (\text{invo}, \ \text{callerCtx}, \ \text{toMeth}, \ \text{calleeCtx}) \\
\leftarrow & \ \\ 
\text{SCall} & \ (\text{toMeth}, \ \text{invo}, \ \text{inMeth}), \\
\text{Reachable} & \ (\text{inMeth}, \ \text{callerCtx}). \\
\text{where} & \ \text{calleeCtx} = \text{MergeStatic}(\text{invo}, \ \text{callerCtx})
\end{align*}
\]

\textit{invo: toMeth(..)}
Rule 6: Virtual Method Calls (VCall)

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).

where calleeCtx = Merge(obj, hctx, invo, callerCtx)
Rule 7: Parameter Passing

\[
\text{InterProcAssign}(\text{to}, \text{calleeCtx}, \text{from}, \text{callerCtx}) \\
\leftarrow \\
\text{CallGraph}(\text{invo}, \text{callerCtx}, \text{meth}, \text{calleeCtx}), \\
\text{ActualArg}(\text{invo}, i, \text{from}), \\
\text{FormalArg}(\text{meth}, i, \text{to}).
\]

\[\text{invo: meth}(...) \rightarrow \text{meth}(..., \text{to}, ..)\]
Rule 8: Return Value Passing

\[
\text{InterProcAssign}(\text{to}, \text{callerCtx}, \text{from}, \text{calleeCtx}) \leftarrow \\
\text{CallGraph}(\text{invo}, \text{callerCtx}, \text{meth}, \text{calleeCtx}), \\
\text{ActualReturn}(\text{invo}, \text{to}), \\
\text{FormalReturn}(\text{meth}, \text{from}).
\]

\textit{invo}: \quad \text{to} = \text{meth}(..) \rightarrow \text{meth}(..) \{ .. \text{return from}; \}
Rule 9: Parameter/Result
Passing as Assignment

\[
\text{VarPointsTo}(\text{to}, \text{toCtx}, \text{obj}, \text{hctx}) \\
\leftarrow \\
\text{InterProcAssign}(\text{to}, \text{toCtx}, \text{from}, \text{fromCtx}), \\
\text{VarPointsTo}(\text{from}, \text{fromCtx}, \text{obj}, \text{hctx}).
\]
Can Now Express Past Analyses Nicely

- 1-call-site-sensitive with context-sensitive heap:
  - Context = HContext = Instr

- Functions:
  - \textit{Record}(\textit{obj}, \textit{ctx}) = \textit{ctx}
  - \textit{Merge}(\textit{obj}, \textit{hctx}, \textit{invo}, \textit{callerCtx}) = \textit{invo}
  - \textit{MergeStatic}(\textit{invo}, \textit{callerCtx}) = \textit{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:**
  - $Context = Iinstr^2$, $HContext = Instr$

- **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 (all with major benefits)

- Discovered that the same name was used for two past algorithms with very 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).
Impressive Performance, Implementation Insights
[OOPSLA’09, ISSTA’09]
Impressive Performance

- Compared to Paddle
  - most complete, scalable past framework
    - includes analyses with a context sensitive heap
- Large speedup for fully equivalent results
  - 15.2x faster for 1-obj, 16.3x faster for 1-call, 7.3x faster for 1-call+heap, 6.6x faster for 1-obj+heap
- Large speedup for more precise results!
  - 9.7x for 1-call, 12.3x for 1-call+heap, 3x for 1-obj+heap
- Scaling to analyses Paddle cannot handle
  - 2-call+1-heap, 2-object+1-heap, 2-call+2-heap
1-call-site-sensitive+heap
Where Is The Magic?

- Surprisingly, 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
Identifying Performance Problems

- Inefficient Datalog:

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

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

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

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

- LogicBlox Datalog indexes on the *last* variable in a relation
- This code joins on the *first* variables of Assign, VarPointsTo

- After re-ordering variables:
More Challenging

- Specification

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

- Semi-naïve executions

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

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

Yannis Smaragdakis
University of Athens
Folding

- **Original**

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

- **Implemented with an extra relation and an extra index**

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

  StoreObjectField(?baseobj, ?field, ?from)
  StoreField(?from, ?field, ?base),
  VarPointsTo(?baseobj, ?base),
Optimization Practice

● Original

\[
\text{VarPointsTo}(\text{?obj}, \text{?to}) \leftarrow \\
\text{LoadField}(\text{?to}, \text{?base}, \text{?field}), \\
\text{VarPointsTo}(\text{?baseobj}, \text{?base}), \\
\text{FieldPointsTo}(\text{?obj}, \text{?field}, \text{?baseobj}).
\]

● Optimize the variable order of LoadField and add folds
  ● You may not change variable order in other relations
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!
Algorithmic Enhancements

- BDDs are necessary if one is not careful about precision
- We introduced simple algorithmic enhancements to avoid redundancy
  - static initializers handled context-insensitively
  - on-the-fly exception handling
- Better analyses, as well as faster!
Conclusions, Future Work
Declarative Program Analysis

- Doop is probably the most complete points-to analysis framework for Java
  - aids exploration
  - competitive performance
- Already yielded new algorithms, implementation techniques
- Latest: flow-sensitive, LLVM, client analyses
- Future: commercialization as program comprehension service