Pride and Prejudice and Zombies

By Jane Austen and Seth Grahame-Smith
Declarative Static Analysis and Zombies

Yannis Smaragdakis
University of Athens

with
Martin Bravenboer,
George Kastrinis, George Balatsouras,
Tony Antoniadis, George Fourtounis, Neville Grech

and
Kostas Ferles, Nikos Filippakis,
Sifis Lagouvardos, Yue Li, Petros Pathoulas,
Kostas Saidis, Tian Tan, Konstantinos Triantafyllou
Declarative Static Analysis and Soundness

Yannis Smaragdakis
University of Athens

with
Martin Bravenboer,
George Kastrinis, George Balatsouras,
Tony Antoniadis, George Fourtounis, Neville Grech

and
Kostas Ferles, Nikos Filippakis,
Sifis Lagouvardos, Yue Li, Petros Pathoulas,
Kostas Saidis, Tian Tan, Konstantinos Triantafyllou
Overview

- **What do we do?**
  - static program analysis
  - “discover program properties that hold for all executions”

- **Vision: a system that knows more about your program than you do**

- **How do we do it?**
  - declarative (logic-based specification)
    - fast, powerful, new insights
Our Research: Doop and friends: CClyzer, MadMax

• Since 2008:
  • Doop: a powerful framework for analyzing Java bytecode
    • building on pointer analysis
      ▪ now just a substrate for more analyses
  • declarative, using the Datalog language

• Lots of offshoots
  • CClyzer, for LLVM bitcode
  • MadMax/Gigahorse for Ethereum VM bytecode
    [OOPSLA'18 Distinguished Paper Award]
      • 38MLoC in 8 hours
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

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

Cloning-based context-sensitive pointer alias analysis using binary decision diagrams
variation points unclear

every variant a new algorithm

correctness unclear

incomparable in precision

Algorithms Found In a 10-Page Pointer Analysis Paper

Figure 1: Example of an algorithm for pointer analysis

Figure 2: Algorithm for computing alias analysis

Figure 3: Algorithm for computing alias analysis

Figure 4: Reintroduce aliases for naive falsification

Figure 5: Reiteration for the incremental algorithm

Figure 6: Procedure for falsifying aliases corresponding to step 1 in Figure 2

Figure 7: Procedures for falsifying aliases which are potentially affected by adding a pointer assignment

Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment
Program Analysis: a Domain of Mutual Recursion

- var points-to
- call graph
- dynamic proxies
- exceptions
- obj fld values
- reflection
Holistic Program Analysis:
“Everything Is Connected”
A Vision Within Reach

• *An intelligent system that knows more about your program than you do*

• “Everything is connected”
  – all analysis aspects encoded separately, all benefitting each other

• The Doop framework serves to illustrate

• Key: a declarative specification of all sorts of static analyses

• In Doop: use of Datalog
Datalog To The Rescue!

- Datalog is relations + recursion
- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - e.g., as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - monotonic

Less programming, more specification
Datalog: Declarative Mutual Recursion

source

```java
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```
Datalog: Declarative Mutual Recursion

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

<table>
<thead>
<tr>
<th>Alloc</th>
<th>Move</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).
```
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(var, obj) <-
Alloc(var, obj).

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

head
**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).`

**head relation**
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()

Move
a   b
b   a
b   c

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

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

body relations
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

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

VarPointsTo

\[ \begin{align*}
  \text{VarPointsTo}(\text{var, obj}) &\leftarrow \text{Alloc}(\text{var, obj}). \\
  \text{VarPointsTo}(\text{to, obj}) &\leftarrow \text{Move}(\text{to, from}), \\
                       &\quad \text{VarPointsTo}(\text{from, obj}).
\end{align*} \]
Datalog: Declarative Mutual Recursion

source

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

Alloc

\[ \text{VarPointsTo} \]
\[ \text{VarPointsTo} \]

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

\[ \text{VarPointsTo}(\text{to}, \text{obj}) \leftarrow \]
\[ \text{Move}(\text{to}, \text{from}), \]
\[ \text{VarPointsTo}(\text{from}, \text{obj}). \]
Datalog: Declarative Mutual Recursion

source

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

Alloc

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

VarPointsTo

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

Move

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

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

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

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

2nd rule evaluation
## 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>a</td>
<td>new B()</td>
</tr>
<tr>
<td>b = a;</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>c = b;</td>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

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

VarPointsTo(to, obj) <- Move(to, from), VarPointsTo(from, obj).
# 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 new A()</td>
<td>a new A()</td>
</tr>
<tr>
<td>b = new B();</td>
<td>b new B()</td>
<td>b new B()</td>
</tr>
<tr>
<td>c = new C();</td>
<td>c new C()</td>
<td>c new C()</td>
</tr>
<tr>
<td>a = b;</td>
<td>a new B()</td>
<td>a new B()</td>
</tr>
<tr>
<td>b = a;</td>
<td>b new A()</td>
<td>b new A()</td>
</tr>
<tr>
<td>c = b;</td>
<td>c new B()</td>
<td>c new B()</td>
</tr>
</tbody>
</table>

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

VarPointsTo(to, obj) <=
Move(to, from),
VarPointsTo(from, obj).
The Doop Framework

• Datalog-based static 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
Past Approaches and Declarative Analysis

- Past approaches have flirted with declarative analysis
- But no purely declarative approach
  - specification and algorithm confused
- Declarativeness considered unscalable in both complexity and performance
  - “the first time I write an analysis it is typically in Datalog, but then, once I’m convinced it’s precise, I throw it out and I write it in Java, when I want to focus on scalability.” (Naik, 2010)
Doop Makes Declarative Analysis Real

- Complete, complex pointer analyses in Datalog
  - core specification: ~1500 logic rules
  - parameterized by a handful of rules per analysis flavor
- 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
  - no BDDs
Not Expected

• Expressed complete, complex pointer analyses in Datalog
  
  “Encoding 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”
Impressive Performance, Implementation Insights

[OOPSLA’09, ISSTA’09]
Large Speedup For Realistic Analyses
Better Understanding of Existing Algorithms, More Precise and Scalable New Algorithms

[PLDI’10, POPL’11, CC’13, PLDI’13, PLDI’14, FSE'18, OOPSLA'18]
Many More Work Threads

- Set-based pre-analysis [OOPSLA’13]
  - universal optimization technique
- Completing a partial program [OOPSLA’13]
  - making sense out of missing libraries
- Soundness [CACM 2/15, ECOOP’18 (distinguished paper)]
- Reflection and dynamic loading [APLAS’15, ECOOP’18, ISSTA’18]
- Port to Souffle: a parallel Datalog engine [SOAP’17]
- Must-alias analysis [SOAP’17, CC’18]
- Taint analysis using points-to algorithms [OOPSLA’17]
- Integrating heap snapshots in static analysis [OOPSLA’17, ISSTA’18]
Now Zombies
(ahem, soundness)
Soundness in Static Analysis

- We all want it!
- **Sound**: \( \text{AnalysisClaim}(P) \rightarrow P \)
- E.g., for a (may-) value-flow analysis: is every possible run-time value modeled statically?
- Soundness is a design property of an analysis
  - often broken up by language feature
    - basically “do you fully handle this feature?”
  - e.g., “do you handle arrays soundly?”
Soundiness Manifesto [CACM 2/15]

• “There is no practical static whole-program may-analysis that is sound”
  − whole-program: models the heap

• What about all these soundness proofs?
  − proof is for a limited language
  − unsoundness due to dynamic features: reflection, dynamic loading, eval

```java
Method m = obj.getClass().getMethod(methName);
m.invoke(obj);
```
This Work: [ECOOP'18, Distinguished Paper Award]

Truly Sound Analysis, for Full Language

- Key elements:
  - I. different form of soundness theorem
  - II. **defensive** design that withstands *opaque* code
    - i.e., code that could be doing (nearly) anything
  - III. laziness necessary for a realistic implementation
Part I. Motivation: Different Form of Soundness Theorem
Conventional Soundness Theorem (formulation by Xavier Rival)

- for all programs in stated language subset and all executions in stated exec. subset
  \[ \text{AnalysisClaim}(P) \rightarrow P \]

- Soundness is always qualified
- Problem: qualifications don't hold in practice
  - realistic programs use dynamic features
Even Worse: Perverse Incentives!

- for all programs in stated language subset and all executions in stated exec. subset
  \[ \text{AnalysisClaim}(P) \rightarrow P \]

- Proof starts from formulation of analysis over input language

- Weaker analysis, easier soundness theorem!
  - vastly unsound analysis: easy soundness proof
Our Soundness Theorem Form

- for all program points, \( \pi \), in *computed* subset, \( \text{AnalysisClaim}_\pi(P) \rightarrow P_\pi \)

- The analysis works for (nearly) all language features, all executions
  - but qualifies which part of its results is guaranteed sound!
Our Soundness Theorem Form

- *for all program points, $\pi$, in computed subset, $\text{AnalysisClaim}_{\pi}(P) \rightarrow P_{\pi}$*

- Important concept: *coverage*
  - how big is the subset of the program for which the analysis is sound
Part II. Approach: Defensive Design that Withstands *Opaque* Code
General Form of Sound Points-To Analysis

• Sound points-to information: need to compute \textit{all} possible values that may \textit{ever} arise at run time

• For the analysis to certify points-to set as sound, it needs to:
  - closely track information \textit{all the way} from its source
  - ensure no possible interference

• Need precise analysis:
  - context-sensitive, flow-sensitive, over access paths
When Can We Be Sound?
Hello-World Case

```java
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

a       new A1()

a       new A2()

a       ???
```
When Can We Be Sound?
Hello-World Case

```java
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;
}
```

program

```

```

points-to

```
a new A1()
a new A2()
a (foo) new A1() + a (bar) new A2()
```
When Can We Be Sound? Hello-World Case

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

```java
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;
}
```
More Illustration

• What do we know after this statement?

  \[ x.fld = \text{new } A(); \quad // \text{abstract object a1} \]

  - that program expression \( x.fld \) refers to \( a1 \)
  - regardless of what \( x \) refers to
    • access paths!
  - also that any \( z.fld \) needs to be augmented

• ... if followed by:

  \[ ... \quad // \text{analyzable code, no interference} \]
  \[ y = x.fld; \]

we know \( y \) also refers to \( a1 \)
Defensiveness Examples

• When the analysis is uncertain, it has to refuse to certify the soundness of a points-to set

```java
if (P()) {
    x.fld = new A();  // abstract object a1
} else {
    x.foo();         // opaque
}
```

• \texttt{x.fld} has an unknown points-to set after \texttt{if}

  - \texttt{x.foo()} could invoke dynamic code, do reflection, or merely be too complex to analyze precisely
    * e.g., reach maximum context-sensitivity depth
Method Calls

• Let's analyze the example further: when is a call **not** opaque code?

```java
if (P()) {
    x.fld = new A();  // abstract object a1
} else {
    x.foo();          // opaque
}
```

- `x` has known points-to set (i.e., known `foo`)
- all possible `foo` do not perform opaque actions on an access path

• Involved topic, more in the paper
Part III. Technique: Laziness for Realistic Implementation
Laziness

- A flow-sensitive, context-sensitive algorithm over access paths cannot scale
- Idea: compute points-to set only when we can prove the set is sound
- Implication: an empty set means unbounded
  - the analysis could not compute all its possible contents
Laziness, Concretely

- All points-to sets start empty
- Only compute a points-to set (i.e., make it non-empty) when
  - all other points-to sets feeding into it are known
  - and are non-empty themselves
- Any points-to set that remains empty at end of analysis is marked T
Laziness Benefits

• Scalable analysis

• Avoids wasted work! Never compute a points-to set, only to have the addition of more information make its contents non-certifiably sound!
  – i.e., T
Evaluation Results
Running Time
Coverage
Devirtualization Client
Conclusions

• Doop: early instance of intelligent system that just *knows* things about your program
• Also: fully sound analysis, for realistic languages, is possible!
• Different form of soundness theorem, *coverage* as important concept