# What's the point?

## Points-to as a Constraint Problem

**Patrick Nappa,**
**Bernhard Scholz,**
**Jens Dietrich**

November 20, 2017

THE UNIVERSITY OF
SYDNEY

# Problem Statement

- Points-To important in programming language tools
  - Compiler
  - Integrated Development Environments
  - Static Program Analysis/Bug Finder
  - Refactoring tools, etc.
- Points-to builds memory-abstraction of heap w/o executing program
- Source-codes become larger, e.g., Linux-Kernel (20+MLOC), OpenJDK
- Problem Statement:
  - How to design scalable points-to algorithms for OO languages

# Andersen Analysis

```
1 int* x,y,z;
2 x = new int();
3 y = new int();
4
5 z = x;
6 z = y;
7 x = y;
```

$$z \mapsto \{o_2, o_3\}$$

$$x \mapsto \{o_2, o_3\}$$

$$y \mapsto \{o_3\}$$

# (some) Classification of Analyses

- **Flow sensitive/insensitive:** Whether we capture the ‘forking’ of control-flow, i.e. does the points-to set represent the flow whether an if condition succeeded or not?
- **Field sensitive/insensitive:** Capturing contents of records (structs) and/or objects..
- **Context sensitive/insensitive:** Whether we capture the call-site information (imagine heap & stack implicit states) in methods.

# Traditional Methods (Background)

- **Set inclusion:** As per before (Andersen)
- **Equality:** Don't bother about subsets on assign, treat assigns as merges
- **Context-free language reachability:** Converting the problem to a grammar, and finding matches
- **Graph conversion:** Similar to above, applying reductions to the graph

# Program Flow

- Translate points-to problem as a constraint problem
- Find new constraint system for describing points-to
- Find scalable solver for constraint system

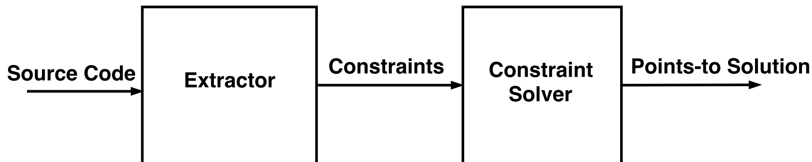


Figure: Points-to Solver

# Primitive Statements for Points-To

- Introduce a simplified set of primitive statements
- Sufficient to capture Java's semantics for points-to
- Flow-insensitive, inclusion-based, field-sensitive
- Primitive Statements:
  - **Object-creation:** $o_k$: `x = new();`
  - **Assignment:** `y := x;`
  - **Load:** `y := p.f;`
  - **Store:** `q.f := x;`

# Load/Store Pair Trick

- Building object-field representation via load/store pairs
  - M. Sridharan, et al OOPSLA'05 via CFL-Reachability
  - Avoids storing set of objects for fields in an object
  - Reduces to a simple variable to object map
  - Space / Computation trade-off
- Replace all load/stores by new conditional statement
- For all load/store pairs whose field is identical:

```
q.f:=x; y:= p.f
```
$\Longrightarrow$
```
if alias(p,q) then y := x;
```

- No notion of object field after replacement
- Conditional alias check for variables required

# Constraint System for Load/Store Semantics

- Define a new constraint system $(C, X, \mathbb{D})$
- Set of constraints, $C = \{c_1, .., c_l\}$.
- Each constraint $c_i, (1 \leq i \leq l)$ is one of the following:

  1. **Subset Constraint:** $x_i \subseteq x_j (i \leq i, j \leq n)$
  2. **Membership Constraint:** $d_j \in x_i (1 \leq i \leq n, 1 \leq j \leq m)$
  3. **Conditional Subset Constraint:** $\phi(x_p) \cap \phi(x_q) \neq \emptyset \Rightarrow x_i \subseteq x_j (1 \leq i, j, p, q \leq n)$

- Set of variables $X = \{x_1, ..., x_n\}$
- Finite domain $\mathbb{D} = \{d_1, ...d_m\}$.

# Solution of Constraint System

- A variable assignment :

$$\phi : X \to 2^{\mathbb{D}}$$

  is a solution of a constraint system $(C, X, \mathbb{D})$
- All constraints $c_i$ hold for a solution.
- A solution is a minimal variable assignment.
- Minimality defined via partial order for solutions

$$\phi_1 \sqsubseteq \phi_2 \Leftrightarrow \forall x \in X : \phi_1(x) \subseteq \phi_2(x)$$

- Infimum: greatest variable assignment $\phi_{inf}$, such that: $\forall i : \phi_{inf} \sqsubseteq \phi_i$

# Translation

- Translation of primitive statements to constraint

| Statement | Constraint |
|---|---|
| `X = new Obj()` | $o_i \in X$ |
| `X := Y` | $Y \subseteq X$ |
| `if alias(P,Q) then Y := X;` | $\phi(P) \cap \phi(Q) \neq \emptyset \Rightarrow Y \subseteq X$ |

# Mechanism

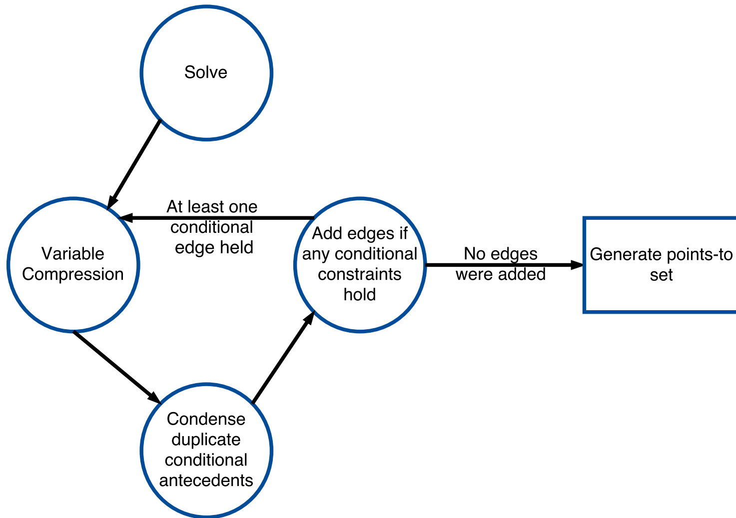


Figure: Architecture of Solver

# Variable Compression

- Define a binary relation $\sim$ on the set $X$:

$$x_1 \sim x_2 \Leftrightarrow \phi(x_1) = \phi(x_2)$$

- Compression of constraint system if binary relation is known a-priori:

$$(C_\sim, [X]_\sim, \mathbb{D})$$

- Can we find $\sim$ faster than $\phi$? ...probably not fully!
- However, we can find approximations, i.e.,

$$x_1 \approx x_2 \Rightarrow x_1 \sim x_2$$

  where $\approx$ is an under-approximation of $\phi$.

# Finding $\approx$ for Compression

- Techniques to find $\approx$:
  - Unreachability, i.e., all unreachable variables of constraint graph in a class
  - Strongly Connected Components of Constraint Graph (Rountev et al. SIGPLAN 2000)
  - Dominance Relation (Nasre ISSM'12)
  - Predecessor Equivalence
- Equivalences are merged from each technique
- Techniques must be fast and effective
- Find-Union data-structures for building equivalence relations over variables
- Relation $\approx$ resembles unified framework to accommodate various techniques

# Compressed Conditional Constraints

- Reduce variable in condition to class representatives
- Example: $y \approx z$

$$C: \begin{array}{l} \phi(x) \cap \phi(y) \neq \emptyset \implies a \subseteq b \\ \phi(x) \cap \phi(z) \neq \emptyset \implies c \subseteq d \end{array}$$

transforms to...

$$C_{\approx} : \phi(x) \cap \phi(\{y, z\}) \neq \emptyset \implies \{a \subseteq b, c \subseteq d\}$$

- 80% reduction(!) of $\widetilde{C}$ on the OpenJDK data set

# Covering problem

Checking whether object load/store pairs alias currently requires calculation of $\phi(x)$ & $\phi(y)$, and checking $\phi(x) \cap \phi(y) \neq \emptyset$

Can it be done another way?

# Covering problem

Checking whether object load/store pairs alias currently requires calculation of $\phi(x)$ & $\phi(y)$, and checking $\phi(x) \cap \phi(y) \neq \emptyset$

Can it be done another way? **Yes.**

# Covering problem

Checking whether object load/store pairs alias currently requires calculation of $\phi(x)$ & $\phi(y)$, and checking $\phi(x) \cap \phi(y) \neq \emptyset$

Can it be done another way? **Yes.**
Is that way faster?

# Covering problem

Checking whether object load/store pairs alias currently requires calculation of $\phi(x)$ & $\phi(y)$, and checking $\phi(x) \cap \phi(y) \neq \emptyset$

Can it be done another way? **Yes.**
Is that way faster? **..maybe?**

# Covering problem

By reflecting along the "clothesline" where all objects hang, we can generate aliases on reachability.

In a single traversal, we can exhaustively find all alias pairs to a given starting node. This can either confirm conditional pairs, or remove candidates.

We may be able to reduce the number of traversals, depending on on how we choose our starting node, and exploiting symmetry. TBD!

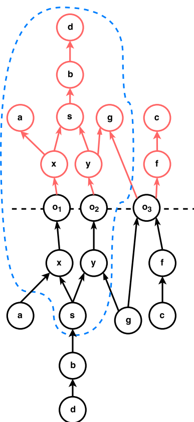


Figure: Traversal along the reflected graph

# Results

Preliminary results for OpenJDK, noting that these categories are not mutually exlusive:

- 1,440,637 variables
- 697,038 variables that are unreachable
- 352,208 variables that are dominated by other variables
- 117,998 quasi-duplicate conditional constraint antecendents, leading to 50% less nodes visited
- Speedups TBD

More optimisations to be had!

Cheers