Who needs better pointer analysis?

IDEs:
- for refactoring, program understanding
- but program edits invalidate analysis
- current analyses too slow to re-analyze
- incremental analyses hard to engineer

JIT compilers:
- for virtual call resolution, register allocation
- but current analyses too slow for runtime
- plus, class loading invalidates, needs re-analysis
What we provide

Analysis so fast that you can
  • run it in the JIT compiler
    • 16x speedup compared to Andersen’s analysis
  • rerun it after the code changes
    • 2ms per query about a pointer variable

Analysis with low memory overhead
  • < 50 KB, eases engineering effort

Analysis with tunable precision
  • adjustable to different time constraints
Contributions

1) Demand analysis with **early termination**:
   - return conservative result after a time out

Problem we had to solve:
   - how to **approximate** to make early termination rare?

2) **Refining** the approximation

Problems we had to solve:
   - **Mechanism**: how to refine?
   - **Policy**: where to focus the refinement budget?
Outline

- Points-to analysis background
- Our approach
  - Demand analysis
  - Early termination
- Our algorithms
  - CFL-reachability formulation
  - Approximation
  - Refinement (undoing the approximation)
- Experiments
Points-To Analysis

- Compute objects each variable can point to
  - For each var x, **points-to set** \( pt(x) \)
- Andersen’s Analysis: our reference point
  - Want similar precision for our analysis
  - One **abstract location** for each allocation site
    \[
    x = \text{new Foo()} \quad \text{yields} \quad pt(x) = \{ o1_{\text{Foo}} \}
    \]
  - Context- and flow-insensitive
- Current implementations not suitable for us
  - Too costly for JIT, IDE
    - 30 s / 30 MB (Berndl et. al. PLDI03) on jedit
  - Code changes require re-analysis
Demand-Driven Analysis

Protocol:
- Client asks a query: what’s the points-to set of variable x?
- Analysis computes only the points-to set of x

Works well when typically few queries:
- JIT compiler: variables in hot code
- IDE: variables in code being edited by developer

Visits theoretically minimal set of statements

Problem:
- worst-case time same as exhaustive
- Happens in practice for standard Andersen’s

Lesson: Need to approximate for scalability
- Ideally, maintain nearly all precision
Approx: Early Termination

Terminate queries when budget exhausted
Return a sound result to client
  • early result: $pt(x) = \{ \text{all abstract locs} \}$
  • complete result: $pt(x) = \{ o_{1\text{Foo}}, o_{2\text{Bar}} \}$

No precision loss if complete result does not satisfy client

Hypothesis: long-running ) unsatisfying
  • Suggested previously (Heintze / Tardieu PLDI01)
  • For standard Andersen’s, large precision loss

Challenge: how to approximate further?
Key Ideas

Formulate analysis in **CFL-reachability**
- Natural for demand-driven analysis
- Andersen’s for Java is **balanced parens**

Approximate through **regularization**
- Solvable by linear DFS algorithm

Iterative refinement to de-approximate
- Simple recursive queries
- Client-driven
CFL-Reachability

Points-to analysis graph:
- Nodes represent variables / locs
- Edges represent statements

Points-to analysis paths:
- o 2 pt(x), *flowsTo*-path from o to x
- pt(x) Å pt(y) ≠ ; , *alias*-path from x to y
Andersen’s Analysis in CFL-Reachability

x = new Obj(); // o₁
z = new Obj(); // o₂
w = x;
y = x;
y.f = z;
v = w.f;

flowsTo ! new (assign)*
gf[f] | assign)*

Field-sensitive formulation: standard for Java
See paper for alias grammar
Approx: Regularization

Add **match edges** for matching field read/write pairs
- From source of putfield
- To sink of getfield

**Regular grammar**
- Yields **DFS algorithm**

**Field-based precision**

\[
\text{flowsToReg} \left( \text{new} \left( \text{pf}[f] \text{ match alias gf}[f] \text{ assign} \right)^* \text{ assign} \right)
\]

\[
\text{pf}[f] \text{ alias gf}[f] \) match
\]

\[
\text{o flowsTo x} \) \text{ o flowsToReg x}
\]
RegularPT

marked, worklist: Set of Node

procedure query(source: Node)
    add source to marked and worklist
while (worklist is non-empty) do
    remove w from worklist
    foreach NEW edge o -> w do
        add o to points-to set of source
    end
    foreach ASSIGN and MATCH edge y -> w do
        if y unmarked, add y to marked and worklist
    end
end
end

• **No caching**, so very low memory usage
• Early termination through traversal budget
Refining Match Edges

Most approximation can be refined

- Imprecise for recursive fields
Client-Driven Refinement Policy

Not clear when / where to refine
- Extra queries may be costly
- Refining match edge may not affect result

**Client-driven: only refine when client affected**
- E.g, multiple targets for virtual call
- Guyer and Lin SAS03

**RefinedRegularPT:**
- Refine edges traversed by RegularPT
- Iterate until client satisfied or budget exhausted
Experimental Hypotheses

1) Algorithms **precise with early termination**
   - Regular approximation reasonable
   - Refinement yields improved precision

2) Algorithms **meet performance goals**
   - Fast running time
   - Low memory
Evaluation Framework

Implemented in Soot / SPARK framework

Benchmarks: SPEC, Ashes, jedit

Clients

- IDE: Virtual call resolution
  - For program understanding
- JIT: queries from hot code
  - Virtual call resolution (for inlining)
  - Local aliasing (for load/store elimination)
## Algorithms

<table>
<thead>
<tr>
<th>Name</th>
<th>Field-based / sensitive</th>
<th>Demand / Exhaustive</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegularPT</td>
<td>Field-based</td>
<td>Demand</td>
<td>DFS Traversal</td>
</tr>
<tr>
<td>RefinedRegularPT</td>
<td>Variable up to partially field-sensitive</td>
<td>Demand</td>
<td>Client-driven refinement</td>
</tr>
<tr>
<td>FullFS</td>
<td>Field-sensitive</td>
<td>Demand</td>
<td>Heintze and Tardieu [PLDI01] adapted to Java</td>
</tr>
<tr>
<td>ExhaustiveFB</td>
<td>Field-based</td>
<td>Exhaustive</td>
<td>from SPARK</td>
</tr>
<tr>
<td>ExhaustiveFS</td>
<td>Field-sensitive</td>
<td>Exhaustive</td>
<td>from SPARK</td>
</tr>
</tbody>
</table>
1) Evaluation: Precision

**jedit virtual calls**

- **RegularPT**: 90% at 50 nodes (2ms)
- **RefinedRegularPT**: 94% at 1250 nodes (20ms)
- **FullFS**: 47.2% resolved at 30,000 nodes (500ms); > 300,000 nodes for 100% (>30s)
2) Evaluation: Performance

javac virtual calls in hot code

ExhaustiveFS: 100% in 16s

ExhaustiveFB: 100% in 7.4s

RegularPT: 100% in 0.49s

FullFS: 50% in 7.4s

Memory:

- <50 KB for (Refined)RegularPT
- 28MB for FullFS using BDDs
Conclusions

New demand points-to analysis

- Speed through **two approximations**
  - Early termination
  - Regularization
- **Refinement driven by client**

Provide **high precision in tight budget**
Suitable for JITs, IDEs; and elsewhere?