Outline

1. **Introduction**
2. **Andersen’s Analysis**
   - The Algorithm
   - Constraints
   - Complexity
3. **Steensgaard’s Analysis**
   - The Algorithm
   - Making it Work
   - Complexity
4. **Comparison**
5. **Hybrids**
Introduction and Rationale

Introduction

- Last time we saw flow sensitive points-to analysis
- Computes information at every point of a program
  - Precise
  - The information is a (large) graph — expensive!

Flow-Insensitive analysis

- Compute just one graph for the entire program
- Consider all statements regardless of control-flow
- SSA or similar forms can recover some precision
Introduction and Rationale

Introduction

- Last time we saw flow sensitive points-to analysis
- Computes information at every point of a program
  - Precise
  - The information is a (large) graph — expensive!

Flow-Insensitive analysis

- Compute just one graph for the entire program
- Consider all statements regardless of control-flow
- SSA or similar forms can recover some precision
A little comparison

**Code**

```c
int x;
int *y, *z;
x = &y;
x = &z;
```

**Flow-Sensitive**

```
x
  -> y

x
  -> z
```

**Flow-Insensitive**

```
x
  -> y

x
  -> z
```

Maks Orlovich  On Flow-Insensitive Points-To Analyses
Outline

1. Introduction
2. Andersen’s Analysis
   - The Algorithm
   - Constraints
   - Complexity
3. Steensgaard’s Analysis
   - The Algorithm
   - Making it Work
   - Complexity
4. Comparison
5. Hybrids
Andersen’s algorithm

- Essentially the immediate adaptation of the usual dataflow points-to algorithm to be flow-insensitive
- Since do not know the order of statements, can say less:
  - \( x = &y \) — can only know that \( y \in pt(x) \)
  - \( x = y \) — can only know that \( pt(y) \subseteq pt(x) \)
- When analyzing, collect such constraints
- Can use a fixed-point computation to compute the actual points-to sets
Constraints for C

1. $x = \& y \leadsto y \in pt(x)$

2. $x = y \leadsto pt(y) \subseteq pt(x)$
Constraints for C II

\[ x = *y \]

\[ \forall a \in pt(y). pt(a) \subseteq pt(x) \]
**Constraints for C III**

\[ *x = y \]

\[ \forall w \in pt(x).pt(y) \subseteq pt(w) \]
Constraints for C — Summary

1. \( x = \& y \quad \Rightarrow \quad y \in pt(x) \)
2. \( x = y \quad \Rightarrow \quad pt(y) \subseteq pt(x) \)
3. \( x = *y \quad \Rightarrow \quad \forall a \in pt(y). pt(a) \subseteq pt(x) \)
4. \( *x = y \quad \Rightarrow \quad \forall w \in pt(x). pt(y) \subseteq pt(w) \)
Constraints for Java

1. Stack variables cannot be pointed to, only heap objects can be.
2. Can take advantage of type safety.
3. The following is one memory abstraction:
   - Name objects by allocation site.
   - Variables point to objects.
   - Fields of objects point to objects.
Constraints for Java II

1. \( x = y \rightarrow \text{pt}(y) \subseteq \text{pt}(x) \)
2. \( y.f = x \rightarrow \forall o \in \text{pt}(y) \ (\text{pt}(x) \subseteq \text{pt}(o.f)) \)
3. \( x = y.f \rightarrow \forall o \in \text{pt}(y) \ (o.f \subseteq \text{pt}(x)) \)
Cost of the algorithm

Asymptotically

- Implicitly have a constraint graph, $O(n)$ nodes, $O(n^2)$ edges
- The fixed point computation essentially computes transitive closure — which is an $O(n^3)$ computation

In practice

- Usually, nowhere near that bad...
- ... but can be bad enough to be unusable
Cost of the algorithm

Asymptotically

- Implicitly have a constraint graph, $O(n)$ nodes, $O(n^2)$ edges
- The fixed point computation essentially computes transitive closure — which is an $O(n^3)$ computation

In practice

- Usually, nowhere near that bad...
- ... but can be bad enough to be unusable
## Actual Performance (from [ShHo97])

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (LoC)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangle</td>
<td>1986</td>
<td>2.9</td>
</tr>
<tr>
<td>gzip</td>
<td>4584</td>
<td>1.7</td>
</tr>
<tr>
<td>li</td>
<td>6054</td>
<td>738.5</td>
</tr>
<tr>
<td>bc</td>
<td>6745</td>
<td>5.5</td>
</tr>
<tr>
<td>less</td>
<td>12152</td>
<td>1.9</td>
</tr>
<tr>
<td>make</td>
<td>15564</td>
<td>260.8</td>
</tr>
<tr>
<td>tar</td>
<td>18585</td>
<td>23.2</td>
</tr>
<tr>
<td>espresso</td>
<td>22050</td>
<td>1373.6</td>
</tr>
<tr>
<td>screen</td>
<td>24300</td>
<td>514.5</td>
</tr>
</tbody>
</table>

75MHz SuperSPARC, 256MB RAM
Reducing the cost

- Cycles in a graph must have the same points-to sets, so can be collapsed to a single node [FäFoSuAi98]
  - In some cases runs at much as 50x faster
  - \texttt{li} is done in 30.25 seconds, \texttt{espresso} in 27 seconds, on UltraSparc in 167-400Mhz
- If two variables have the same points-to sets, they can be collapsed [RoCh00]
  - Around 2x improvement in run time, 3x lower memory usage
- BDDs (Reduced Ordered Binary Decision Diagrams) have been used to represent the graph more sparsely [BeLhQiHeUm03]
Outline

1 Introduction
2 Andersen’s Analysis
   - The Algorithm
   - Constraints
   - Complexity
3 Steensgaard’s Analysis
   - The Algorithm
   - Making it Work
   - Complexity
4 Comparison
5 Hybrids
Overview

- Can view the problem as trying to assign synthetic types to each reference — so it points to objects of specified type
- A type is defined recursively as pointing to another type
- Hence, proceeds as a type inference algorithm, doing unification
- $x = y \rightarrow \tau(x) = \tau(y)$, so take pt($x$) = pt($y$)
- Each type points to one other type, so the points-to graph has at most 1 out edge for each node (but each node can be many variables)
  - Graph is of linear size — fast!
  - Limits precision
Processing of assignments

\[ x = *y \]

\[
\begin{array}{c}
  x \\
  y
\end{array}
\quad \rightarrow 
\begin{array}{c}
  z \\
  w
\end{array}
\quad \rightarrow 
\begin{array}{c}
  a \\
  v
\end{array}
\quad \rightarrow 
\begin{array}{c}
  b
\end{array}
\]
Processing of assignments

\[ x = *y \]

\[
\begin{array}{ccc}
  x & \rightarrow & z & \rightarrow & a \\
  y & \rightarrow & w & \rightarrow & v & \rightarrow & b \\
\end{array}
\]
Processing of assignments

\[ x = *y \]

\[ x \rightarrow z \rightarrow a \]

\[ y \rightarrow w \rightarrow v \rightarrow b \]
Processing of assignments

\[ x = \ast y \]

\[ x \rightarrow z \rightarrow a \]
\[ y \rightarrow w \rightarrow v \rightarrow b \]
Making it work

There are a couple of problems that arise in practice

- Building a call graph
  - Make the type a pair, including a function pointer portion
  - Compute the set of functions that may point to using unification as well

- Integer assignments to pointers/lack of type safety
  - int* a = 0, *x = a, *y = b;
  - Will collapse them into a single node
  - Should only do unification if RHS is known to be a pointer
    - Don’t unify if we don’t see the RHS pointing to anything, just record an edge
    - Perform a unification if RHS gets to point to something
Complexity

It’s fast!

- Asymptotically, $O(N^\alpha(N, N))$
- Has been shown to analyze programs with millions of lines of code in under a minute
Outline

1 Introduction

2 Andersen’s Analysis
   - The Algorithm
   - Constraints
   - Complexity

3 Steensgaard’s Analysis
   - The Algorithm
   - Making it Work
   - Complexity

4 Comparison

5 Hybrids
### Some numbers — time (from [ShHo97])

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (LoC)</th>
<th>Andersen (sec)</th>
<th>Steensgaard (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangle</td>
<td>1986</td>
<td>2.9</td>
<td>0.8</td>
</tr>
<tr>
<td>gzip</td>
<td>4584</td>
<td>1.7</td>
<td>1.1</td>
</tr>
<tr>
<td>li</td>
<td>6054</td>
<td>738.5</td>
<td>4.7</td>
</tr>
<tr>
<td>bc</td>
<td>6745</td>
<td>5.5</td>
<td>1.6</td>
</tr>
<tr>
<td>less</td>
<td>12152</td>
<td>1.9</td>
<td>1.5</td>
</tr>
<tr>
<td>make</td>
<td>15564</td>
<td>260.8</td>
<td>6.1</td>
</tr>
<tr>
<td>tar</td>
<td>18585</td>
<td>23.2</td>
<td>3.6</td>
</tr>
<tr>
<td>espresso</td>
<td>22050</td>
<td>1373.6</td>
<td>10.2</td>
</tr>
<tr>
<td>screen</td>
<td>24300</td>
<td>514.5</td>
<td>10.1</td>
</tr>
</tbody>
</table>

75MHz SuperSPARC, 256MB RAM
### Some numbers — Average alias set size (from [ShHo97])

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (LoC)</th>
<th>Andersen</th>
<th>Steensgaard</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangle</td>
<td>1986</td>
<td>4.01</td>
<td>21.93</td>
</tr>
<tr>
<td>gzip</td>
<td>4584</td>
<td>2.96</td>
<td>25.17</td>
</tr>
<tr>
<td>li</td>
<td>6054</td>
<td>171.14</td>
<td>457.89</td>
</tr>
<tr>
<td>bc</td>
<td>6745</td>
<td>18.57</td>
<td>83.55</td>
</tr>
<tr>
<td>less</td>
<td>12152</td>
<td>7.11</td>
<td>63.75</td>
</tr>
<tr>
<td>make</td>
<td>15564</td>
<td>74.70</td>
<td>414.03</td>
</tr>
<tr>
<td>tar</td>
<td>18585</td>
<td>17.41</td>
<td>53.7</td>
</tr>
<tr>
<td>espresso</td>
<td>22050</td>
<td>109.53</td>
<td>143.4</td>
</tr>
<tr>
<td>screen</td>
<td>24300</td>
<td>106.89</td>
<td>652.8</td>
</tr>
</tbody>
</table>
Relation between algorithms

- Andersen’s algorithm can be viewed as type-inference, too
  - But with subtyping
  - $x = y: \tau(y) \subsetneq \tau(x)$, so $pt(y) \subseteq pt(x)$.
- Steensgaard’s algorithm can be thought as restricting the out-degree of the graph produced by Andersen’s algorithm to 1, by merging nodes when that is exceeded.
Outline

1. Introduction
2. Andersen’s Analysis
   - The Algorithm
   - Constraints
   - Complexity
3. Steensgaard’s Analysis
   - The Algorithm
   - Making it Work
   - Complexity
4. Comparison
5. Hybrids
$k$-limiting

[ShHo97] provides a $k$-limiting algorithm, which with $k = 1$ behave as Steensgaard, with $k = N$ as Andersen

- Assign variables $k$ colors
- Have a separate points-to slot for each color
- Do a few runs with different assignments, and intersect the results ($k^2 \log_k N$ factor slowdown)
- Average alias set size was shrunk by about 1.78,
- About 2x faster than Andersen when that runs slowly, but often slower than it — very high constant factors
One level flow

[Das2000] introduced an another heuristic.

Algorithm

- Observation: C programs mostly use pointers to pass in parameters, which are basically assignments.
- Solution: Accurately model the simple cases by using containment constraints to refer to points-to sets of symbols in the assignment, but unify stuff further out.
- Can get some context sensitivity on top of it, by labeling edges, and doing CFL reachability (makes it $O(n^3)$).
Accuracy

- Produces nearly identical sets as Andersen for most test programs (except one that used pointers to pointers)

Performance

- Asymptotically: linear memory use, quadratic time (in the constraint-solving phase)
- About 2x slower than Steensgaard’s algorithm in practice
- Analyzes 1.4 million lines of code (Word97) in about 2 minutes on a 450Mhz Xeon
Discussion...