Points-To Analysis

Derek Rayside
MIT CSAIL 6.883, Prof Ernst
drayside@mit.edu
November 14, 2005

Contents

1 Overview 1
  1.1 Complexity . . . . . . . . . . . . . . . . . . 1
  1.2 Axes of Precision . . . . . . . . . . . . . . 2

2 Steensgaard [31] Example 3
  2.1 Intuitive formulation [29] . . . . . . . . . . 3
  2.2 Type-based formulation [6] . . . . . . . . . . 3

3 Andersen [1] Example [29] 4

1 Overview

Classic Research Challenge Getting precision for large programs quickly. Recently groups from McGill [2, 20] and Stanford [34, 35] have used binary decision diagrams (BDDs) to make precise analyses scale to large programs. Raman [28] has a brief overview of BDDs in this context.

New Research Challenges Incremental analyses. Incomplete programs. Demand-driven analyses (eg, [30]). Dynamic class loading (eg, [15, 24]).

Software Engineering Decision How to choose the right points-to analysis for the software engineering problem you’re trying to solve. What costs are worth paying? [13] [14] [23] [10]

Liang et al. [22] found Andersen-style (inclusion) analyses significantly more precise than Steensgaard-style (unification).

Lhoták and Hendren [21] found object-sensitivity Milanova et al. [25] gave the most bang for the buck for Java, vs approaches such as [34, 35].

Surveys Hind [12] Raman [28]

Dynamic Analysis Relatively little work done here. Gross [10], Mock et al. [26] show 100x improvement over static analyses, with 100x slowdown in program execution.

Context Sensitivity \{ fun string \}

1.1 Complexity

Abstract from Chakaravarthy [3]:

Given a program and two variables p and q, the goal of points-to analysis is to check if p can point to q in some execution of the program. This well-studied problem plays a crucial role in compiler optimization. The problem is known to be undecidable when dynamic memory is allowed. But the result is known only when variables are allowed to be structures. We extend the result to show that, the problem remains undecidable, even when only scalar variables are allowed. Our second result deals with a version of points-to analysis called flow-insensitive analysis, where one ignores the control flow of the program and assumes that the statements can be executed in any order. The problem is known to be NP-Hard, even when dynamic memory is not allowed and variables are scalar. We show that when the variables are further restricted to have well-defined data types, the problem is in P. The corresponding flow-sensitive version, even with further restrictions, is known to be PSPACE-Complete. Thus, our result gives some theoretical evidence that flow-insensitive analysis is easier than flow-sensitive analysis. Moreover, while most variations of the points-to analysis are known to be computationally hard, our result gives a rare instance of a non-trivial points-to problem solvable in polynomial time.

Ramalingam [27]: Aliasing is undecidable
Landi [17]: PSPACE-complete even with no procedures or memory allocations
Landi and Ryder [18]
Figure 1 A Brief History of Pointer Analysis [33] — focus on scalability and precision

<table>
<thead>
<tr>
<th></th>
<th>Equality-based</th>
<th>Subset-based</th>
<th>Flow-sensitive</th>
</tr>
</thead>
</table>

Horwitz [16]: Even flow-insensitive problem is NP-hard
Chakaravarthy [3]: Cannot even get a good approximation (within a constant factor) unless P=NP

### 1.2 Axes of Precision

<table>
<thead>
<tr>
<th>less precise</th>
<th>more precise</th>
</tr>
</thead>
<tbody>
<tr>
<td>equivalence</td>
<td>subset/inclusion</td>
</tr>
<tr>
<td>flow-insensitive</td>
<td>flow-sensitive</td>
</tr>
<tr>
<td>context-insensitive</td>
<td>context-sensitive</td>
</tr>
</tbody>
</table>

Consider the following example [33]:

```c
p = malloc();
q = malloc();
fp = &p;
fp = &q;
... = *fp;
```

What does the points-to graph look like at the end of the snippet? Depends on what analysis you do:

- flow-insensitive, equality-based, eg Steensgaard [31]

```
flow-insensitive, equality-based, eg Steensgaard [31]
```

- flow-insensitive, subset-based, eg Andersen [1]

```
flow-insensitive, subset-based, eg Andersen [1]
```

- flow-sensitive

```
flow-sensitive
```

Another example, for context-sensitivity [33]:

```c
id (x) { return x; }
foo() {
    a = malloc();
    a = id(a);
}
bar() {
    b = malloc();
    b = id(b);
}
```

```
\begin{figure}
\centering
\begin{tikzpicture}
\node (a) at (0,0) {a};
\node (b) at (1,0) {b};
\node (heap1) at (0,1) {heap1};
\node (heap2) at (1,1) {heap2};
\draw[->] (a) -- (heap1);
\draw[->] (b) -- (heap2);
\end{tikzpicture}
\caption{Dotted lines are spurious edges added by context-insensitivity.}
\end{figure}
```
2 Steensgaard [31] Example

Consider the following program:

1. \( x = \&a; \)
2. \( y = \&b; \)
3. \( p = \&x; \)
4. \( p = \&y; \)

2.1 Intuitive formulation [29]

Now we’ll construct the points-to graph for this program using the Steensgaard approach as formulated by Ryder [29].

First three statements are easy:

\[
\begin{align*}
\text{x = \&a;} & \quad x \rightarrow a \\
\text{y = \&b;} & \quad x \rightarrow a \\
\text{p = \&x;} & \quad y \rightarrow b
\end{align*}
\]

Last statement takes more effort to process:

\[
\begin{align*}
\text{p = \&y;} & \quad p \rightarrow x \rightarrow a \\
\text{add edge} & \quad y \rightarrow b \\
\text{collapse x and y} & \quad p \rightarrow x \rightarrow a \\
\text{collapse a and b} & \quad y \rightarrow b
\end{align*}
\]

Resulting points-to graph over-approximates:

\[
\begin{align*}
\text{points-to graph} & \quad p \rightarrow x \rightarrow a \\
& \quad y \rightarrow b
\end{align*}
\]

Why do we have to do this collapsing? It seems that the analysis would be linear in the size of the program if we didn’t do collapsing. The issues is statements like \( a=b; \) see the example of Andersen’s analysis below for why these introduce more complexity.

2.2 Type-based formulation [6]

First we assign each variable its own type:

\[
\begin{align*}
\text{x : t_1} & \\
\text{y : t_2} & \\
\text{a : t_3} & \\
\text{b : t_4} & \\
\text{p : t_5}
\end{align*}
\]

Then we construct the initial constraints:

\[
\begin{align*}
1. & \quad x = \&a; \quad t_1 = \text{ref}(t_3 \times \_)) \\
2. & \quad y = \&b; \quad t_2 = \text{ref}(t_4 \times \_)) \\
3. & \quad p = \&x; \quad t_5 = \text{ref}(t_1 \times \_)) \\
4. & \quad p = \&y; \quad t_5 = \text{ref}(t_2 \times \_))
\end{align*}
\]

Now we solve/unify the constraints. First we see:

\[
\begin{align*}
& \quad t_5 = \text{ref}(t_1 \times \_)) = \text{ref}(t_2 \times \_))
\end{align*}
\]

So we merge \( t_1 \) and \( t_2 \) into \( t_1 \). The world looks like this:

\[
\begin{align*}
\text{x : t_1} & \\
\text{y : t_1} & \\
\text{a : t_3} & \\
\text{b : t_4} & \\
\text{p : t_5} & \\
& \quad t_1 = \text{ref}(t_3 \times \_)) \\
& \quad t_1 = \text{ref}(t_4 \times \_)) \\
& \quad t_5 = \text{ref}(t_1 \times \_))
\end{align*}
\]

Next we see:

\[
\begin{align*}
& \quad t_1 = \text{ref}(t_3 \times \_)) = \text{ref}(t_4 \times \_))
\end{align*}
\]

So we merge \( t_3 \) and \( t_4 \) into \( t_3 \). The world looks like this:

\[
\begin{align*}
\text{x : t_1} & \\
\text{y : t_1} & \\
\text{a : t_3} & \\
\text{b : t_3} & \\
\text{p : t_5} & \\
& \quad t_1 = \text{ref}(t_3 \times \_)) \\
& \quad t_5 = \text{ref}(t_1 \times \_))
\end{align*}
\]

We’re done solving. The storage shape graph is:

\[
\begin{align*}
& \quad t_5 \rightarrow t_1 \rightarrow t_3
\end{align*}
\]

If we expand that to a points-to graph we get:

\[
\begin{align*}
\text{p : t_5} & \\
\text{x : t_1} & \\
\text{y : t_1} & \\
& \quad a : t_3 \\
& \quad b : t_3
\end{align*}
\]
3 Andersen [1] Example [29]

Consider the following program:

1. \( q = &x; \)
2. \( q = &y; \)
3. \( p = q; \)
4. \( q = &z; \)

First two statements are easy:

\[
\begin{align*}
q &= &x; \\
& 1 & x \\
\end{align*}
\]

\[
\begin{align*}
q &= &y; \\
& 1 & x \\
& 2 & y \\
\end{align*}
\]

Third statement. See all the things \( q \) points to, and make \( p \) point to them as well. Add in dotted line, to remind us \( \text{pts}(q) \subseteq \text{pts}(p) \).

\[
\begin{align*}
p &= q; \\
& 1 & x \\
& 2 & y \\
& 3 & p \\
\end{align*}
\]

Fourth statement. Add in \( q \rightarrow z \) edge.

\[
\begin{align*}
q &= &z; \\
& 1 & x \\
& 2 & y \\
& 3 & p \\
& 4 & z \\
\end{align*}
\]

But dotted line reminds us that \( \text{pts}(q) \subseteq \text{pts}(p) \). So we need to add \( p \rightarrow z \) edge as well. This is the extra work that makes Andersen’s analysis more expensive. In a Steensgaard style analysis we would have collapsed \( x \) and \( y \) at the second statement, and then we wouldn’t have to worry about this extra work (although we would lose precision).

\[
\begin{align*}
q &= &z; \\
& 1 & x \\
& 2 & y \\
& 3 & p \\
& 4 & z \\
\end{align*}
\]

Andersen is \( O(n^3) \).

Steensgaard is said to be equality-based, eg: \( \text{pts}(q) = \text{pts}(p) \).

Acknowledgements

Thanks to Greg Dennis and Rob Seater for discussions. Thanks to John Whaley for sending me his slides [33]. Thanks to Michael Ernst for sending me to Dagstuhl where I saw Barbara Ryder’s talk [29].

References


[12] MICHAEL HIND. Pointer analysis: haven’t we solved this problem yet? In Field and Snelting [9], pages 54–61.


[22] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In Field and Snelting [9], pages 73–79.


