Pointer Analysis

G. Ramalingam
Microsoft Research, India
&
K. V. Raghavan
Goals

• **Points-to Analysis**: Determine the set of possible values of a pointer-variable (at different points in a program)
  – what locations can a pointer point-to?

• **Alias Analysis**: Determine if two pointer-variables may point to the same location

• **Compute conservative approximation**

• **A fundamental analysis, required by most other static analyses**
A Constant Propagation Example

\[x = 3;\]
\[y = 4;\]
\[z = x + 5;\]

- \(x\) is always 3 here
- can replace \(x\) by 3
- and replace \(x + 5\) by 8
- and so on
A Constant Propagation Example
With Pointers

\[ x = 3; \]
\[ *p = 4; \]
\[ z = x + 5; \]

• Is \( x \) always 3 here?
A Constant Propagation Example With Pointers

\[
p = &y; \\
x = 3; \\
*p = 4; \\
z = \boxed{x} + 5;
\]

\[
\text{if (?)} \\
p = &x; \\
\text{else} \\
p = &y; \\
x = 3; \\
\]

\[
p = &x; \\
x = 3; \\
*p = 4; \\
z = \boxed{x} + 5;
\]

- \(x\) is always 3
- \(x\) may be 3 or 4 (i.e., \(x\) is unknown in our lattice)
- Pointers affect most program analyses
- Always 4
A Constant Propagation Example

With Pointers

\( p = &y; \)
\( x = 3; \)
\( p\) always points-to \( y \)
\( *p = 4; \)
\( z = x + 5; \)

if (?)
\( p = &x; \)
else
\( p = &y; \)
\( x = 3; \)
\( *p = 4; \)
\( z = x + 5; \)

\( p\) may point-to \( x \) or \( y \)

\( p = &x; \)
\( x = 3; \)
\( *p = 4; \)
\( z = x + 5; \)

\( p\) always points-to \( x \)
Points-to Analysis

• Determine the set of targets a pointer variable could point-to (at different points in the program)
  - “p points-to x”
    • “p stores the value &x”
    • “*p denotes the location x”
  - targets could be variables or locations in the heap (dynamic memory allocation)
    • p = &x;
    • p = new Foo(); or p = malloc (...);
Algorithm A (may points-to analysis)
A Simple Example

```
p = &x;
q = &y;
if (?) {
    q = p;
}
x = &a;
y = &b;
z = *q;
```
Algorithm A (may points-to analysis)
A Simple Example

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = &amp;x;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q = &amp;y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (?) {</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q = p;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = &amp;a;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y = &amp;b;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z = *q;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm A (may points-to analysis) - A Simple Example

```c
x = &a;
y = &b;
if (?) {
p = &x;
} else {
p = &y;
}
x = &c;
p = &c;
```

How should we handle this statement? *(Try it!)*

- **Strong update**
  - `x: {a,c}`
  - `y: {b,c}`
  - `p: {x,y} a: c`

- **Weak update**
  - `x: a` (strong update)
  - `y: b` (strong update)
  - `p: {x,y} null`
Questions

• When is it correct to use a strong update? A weak update?

• Is this points-to analysis precise?

• We must formally define what we want to compute before we can answer many such questions
Points-To Analysis: An Informal Definition

• Let $u$ denote a program-point

• Define $\text{IdealMayPT}(u)$ to be
  \[ \{(p,x) \mid p \text{ points-to } x \text{ in some state at } u \text{ in some run}\} \]

• Algorithm should compute a set $\text{MayPT}(u)$ that over-approximates above set
Static Program Analysis

- A static program analysis computes approximate information about the runtime behavior of a given program:
  1. The set of valid programs is defined by the programming language syntax.
  2. The runtime behavior of a given program is defined by the programming language semantics.
  3. The analysis problem defines what information is desired.
  4. The analysis algorithm determines what approximation to make.
Programming Language: Syntax

• A program consists of
  – a set of variables \( \text{Var} \)
  – a directed graph \((V,E,\text{entry})\) with a
distinguished entry vertex, with every edge
labelled by a primitive statement

• A primitive statement is of the form
  • \( x = \text{null} \)
  • \( x = y \)
  • \( x = *y \)
  • \( x = &y; \)
  • \( *x = y \)
  • \text{skip} \\

(\text{where } x \text{ and } y \text{ are variables in } \text{Var})
Example Program

```c
x = &a;
y = &b;
if (?) {
    p = &x;
} else {
    p = &y;
}
*x = &c;
*p = &c;

Vars = \{x, y, p, a, b, c\}
```
Programming Language: Operational Semantics

• Operational semantics == an interpreter (defined mathematically)

• State
  – DataState ::= Var -> (Var U {null})
  – PC ::= V (the vertex set of the CFG)
  – ProgramState ::= PC x DataState

• Initial state:
  – (entry, \x. null)
Example States

\[ \text{Vars} = \{x, y, p, a, b, c\} \]

1. \[ x = \&a \]
2. \[ y = \&b \]
3. \[ p = \&x \]
4. \[ p = \&y \]
5. \[ \text{skip} \]
6. \[ \text{skip} \]
7. \[ \ast x = \&c \]
8. \[ \ast p = \&c \]

**Initial data-state**

\[ x: \text{N}, y: \text{N}, p: \text{N}, a: \text{N}, b: \text{N}, c: \text{N} \]

**Initial program-state**

\[ <1, x: \text{N}, y: \text{N}, p: \text{N}, a: \text{N}, b: \text{N}, c: \text{N}> \]

**Next program-state**

\[ <2, x: a, y: \text{N}, p: \text{N}, a: \text{N}, b: \text{N}, c: \text{N}> \]
Programming Language: Operational Semantics

• Meaning of primitive statements
  – $CS[\text{stmt}] : \text{DataState} \rightarrow 2^{\text{DataState}}$

• $CS[ x = \text{null} ] s = \{s[x \rightarrow \text{null}]\}$
• $CS[ x = \&y ] s = \{s[x \rightarrow y]\}$
• $CS[ x = y ] s = \{s[x \rightarrow s(y)]\}$
• $CS[ x = \ast y ] s = \ldots$

  ... 

  ... 

• $CS[\ast x = y ] s = \ldots$

  \ldots 

  = \ldots
Programming Language: Operational Semantics

- **Meaning of primitive statements**
  - \( CS[\text{stmt}] : \text{DataState} \rightarrow 2^{\text{DataState}} \)

- \( CS[ x = \text{null} ] s = \{ s[x \rightarrow \text{null}] \} \)
- \( CS[ x = &y ] s = \{ s[x \rightarrow y] \} \)
- \( CS[ x = y ] s = \{ s[x \rightarrow s(y)] \} \)
- \( CS[ x = \ast y ] s = \{ s[x \rightarrow s(s(y))] \} \),
  - if \( s(y) \) is not null
  - \( = \{ \}, \) otherwise
- \( CS[ \ast x = y ] s = \ldots \)
  - \( \ldots \)
  - \( \ldots \)
Programming Language: Operational Semantics

- Meaning of primitive statements
  - $CS[\text{stmt}] : \text{DataState} \rightarrow 2^{\text{DataState}}$

- $CS[\ x = \text{null} \ ] \ s = \{s[x \rightarrow \text{null}]\}$
- $CS[\ x = \& y \ ] \ s = \{s[x \rightarrow y]\}$
- $CS[\ x = y \ ] \ s = \{s[x \rightarrow s(y)]\}$
- $CS[\ x = * y \ ] \ s = \{s[x \rightarrow s(s(y))]\}$,
  \hspace{1cm} \text{if } s(y) \text{ is not null}
  \hspace{1cm} = \{\}, \text{otherwise}$
- $CS[\ * x = y \ ] \ s = \{s[s(x) \rightarrow s(y)]\}$,
  \hspace{1cm} \text{if } s(x) \text{ is not null}
  \hspace{1cm} = \{\}, \text{otherwise}$
Programming Language: Operational Semantics

• Meaning of program
  – a transition relation $\rightarrow$ on program states
  – $\rightarrow \subseteq \text{ProgramState} \times \text{ProgramState}$
  – $\text{state}_1 \rightarrow \text{state}_2$ means that the execution of some edge in the program can transform $\text{state}_1$ into $\text{state}_2$

• Defining $\rightarrow$
  – $(u,s) \rightarrow (v,s')$ iff the program contains a control-flow edge $u \rightarrow v$ labelled with a statement $\text{stmt}$ such that $\text{CS}[\text{stmt}]s = s'$
Programming Language: Operational Semantics

• A sequence of states \( s_1 s_2 \ldots s_n \) is said to be an execution (of the program) iff
  - \( s_1 \) is the Initial-State
  - \( s_i \rightarrow s_{i+1} \) for \( 1 \leq i < n \)

• A state \( s \) is said to be a reachable state iff there exists some execution \( s_1 s_2 \ldots s_n \) is such that \( s_n = s \).

• Define \( RS(u) = \{ s \mid (u,s) \text{ is reachable} \} \)
Programming Language: Operational Semantics

- A sequence of states $s_1, s_2, \ldots, s_n$ is said to be an execution (of the program) if:
  - $s_1$ is the Initial-State
  - $s_i \rightarrow s_{i+1}$ for $1 \leq i < n$
- A state $s$ is said to be a reachable state iff there exists some execution $s_1 s_2 \ldots s_n$ such that $s_n = s$.
- Define $RS(u) = \{ s \mid (u,s) \text{ is reachable} \}$

This is the collecting semantics at point $u$.
Ideal Points-To Analysis: Formal Definition

• Let $u$ denote a vertex in the CFG

• Define $\text{IdealMayPT}(u)$ to be

$$\\{ x \mid \text{exists } s \in \text{RS}(u). \ s(p) = x \}$$
May-Point-To Analysis: Problem statement

Compute MayPT: $V \rightarrow 2^{\text{Var'}}$ such that for every vertex $u$
$\text{MayPT}(u) \supseteq \text{IdealMayPT}(u)$
(where Var’ = Var U {null})
May-Point-To Algorithms

Compute MayPT: $V \rightarrow 2^{\text{Vars'}}$ such that

$$\text{MayPT}(u) \supseteq \text{IdealMayPT}(u)$$

- An algorithm is said to be **correct** if the solution MayPT it computes satisfies
  $$\forall u \in V. \text{MayPT}(u) \supseteq \text{IdealMayPT}(u)$$

- An algorithm is said to be **precise** if the solution MayPT it computes satisfies
  $$\forall u \in V. \text{MayPT}(u) = \text{IdealMayPT}(u)$$

- An algorithm that computes a solution MayPT1 is said to be **more precise** than one that computes a solution MayPT2 if
  $$\forall u \in V. \text{MayPT1}(u) \subseteq \text{MayPT2}(u)$$
Algorithm A: A Formal Definition
The “Data Flow Analysis” Recipe

• Define semi-lattice of abstract-values
  – AbsDataState ::= (Var -> (2^Var' – {}) ) U {bot}
  – f_1 U f_2 = \forall x. (f_1(x) U f_2(x))

• Define initial abstract-value
  – InitialAbsState = \forall x. {null}

• Define transformers for primitive statements
  • AS[stmt] : AbsDataState -> AbsDataState
Algorithm A: A Formal Definition
The "Data Flow Analysis" Recipe

• Let $st(v,u)$ denote stmt on edge $v \rightarrow u$

  $x(v)$  $x(w)$

  $v$  $w$

  $st(v,u)$  $st(w,u)$

  $u$

  $x(u)$

• Compute the least-fixed-point of the following "dataflow equations"
  $- x(entry) = \text{InitialAbsState}$
  $- x(u) = \bigcup_{v \rightarrow u} \text{AS}(st(v,u)) \cdot x(v)$
Algorithm A: The Transformers

• Abstract transformers for primitive statements
  – AS[stmt] : AbsDataState → AbsDataState

  • AS[ \( x = y \) ] \( s = s[x \rightarrow s(y)] \)
  • AS[ \( x = \text{null} \) ] \( s = s[x \rightarrow \{\text{null}\}] \)
  • AS[ \( x = \& y \) ] \( s = s[x \rightarrow \{y\}] \)
  • AS[ \( x = * y \) ] \( s = s[x \rightarrow s^*(s(y) - \{\text{null}\})] \),
    if \( s(y) \) is not = \{null\}
    = \text{bot}, otherwise

where \( s^*(\{v_1, \ldots, v_n\}) = s(v_1) \cup \ldots \cup s(v_n) \),
Algorithm A

- \text{AS}[ \*x = y ] s =

\begin{align*}
\begin{cases}
\text{bot} & \text{if } s(x) = \{\text{null}\} \\
 s[z \rightarrow s(y)] & \text{if } s(x) - \{\text{null}\} = \{z\} \\
 s[z_1 \rightarrow s(z_1) \cup s(y)] & \text{if } s(x) - \{\text{null}\} = \{z_1, \ldots, z_k\} \\
 [z_2 \rightarrow s(z_2) \cup s(y)] & (\text{where } k > 1) \\
 \vdots \\
 [z_k \rightarrow s(z_k) \cup s(y)]
\end{cases}
\end{align*}

- After fix-point solution is obtained, \text{AbsDataState}(u) is emitted as \text{MayPT}(u), for each program point \text{u}
An alternative algorithm: must points-to analysis

- AbsDataState is modified, as follows:
  - Each var is mapped to {} or to a singleton set
  - join is point-wise intersection

- Let MustPT(u) be fix-point at u

- Guarantee: \( \Upsilon(\text{MustPT}(u)) \supseteq \text{MayPT}(u) \supseteq \text{IdealMayPT}(u) \)

where \( \Upsilon(S) = S \),

\[ \text{if } S \text{ is a singleton set} \]

\[ = \text{Var'}, \text{ if } S = {} \]
Must points-to analysis algorithm

- **AS transfer functions same as in Algorithm A for** \( x = y, x = \text{null}, \text{ and } x = \&y \)

- **AS** \[ x = *y \] \( \rightarrow \) \( s \)

  \[
  = \text{bot}, \quad \text{if } s(y) = \{\text{null}\}
  
  = s[x \rightarrow \{\}\], \quad \text{if } s(y) = \{
  
  = s[x \rightarrow s(z)], \quad \text{if } s(y) = \{z\}
  \]
Must points-to analysis algorithm

• AS[ *x = y ] s = bot,
  
  if s(x) = {null}  
  = s[z \rightarrow s(y)]  
  if s(x) = \{z\}  
  = \forall. \{}  
  otherwise

This analysis is less precise than the may-points-to analysis (Algorithm A), but is more efficient