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 = x + 5;
```

x is alway

```
if (?)
  p = &x;
else
  p = &y;
x = 3;
```

```
p = &x;
x = 3;
*p = 4;
z = x + 5;
```

always 4

pointers affect most program analyses

x may be 3 or 4 (i.e., x is unknown in our lattice)

A Constant Propagation Example With Pointers

```
p = &y;
x = 3;
*p = 4;
z = x + 5;
```

p always points-to y

```
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

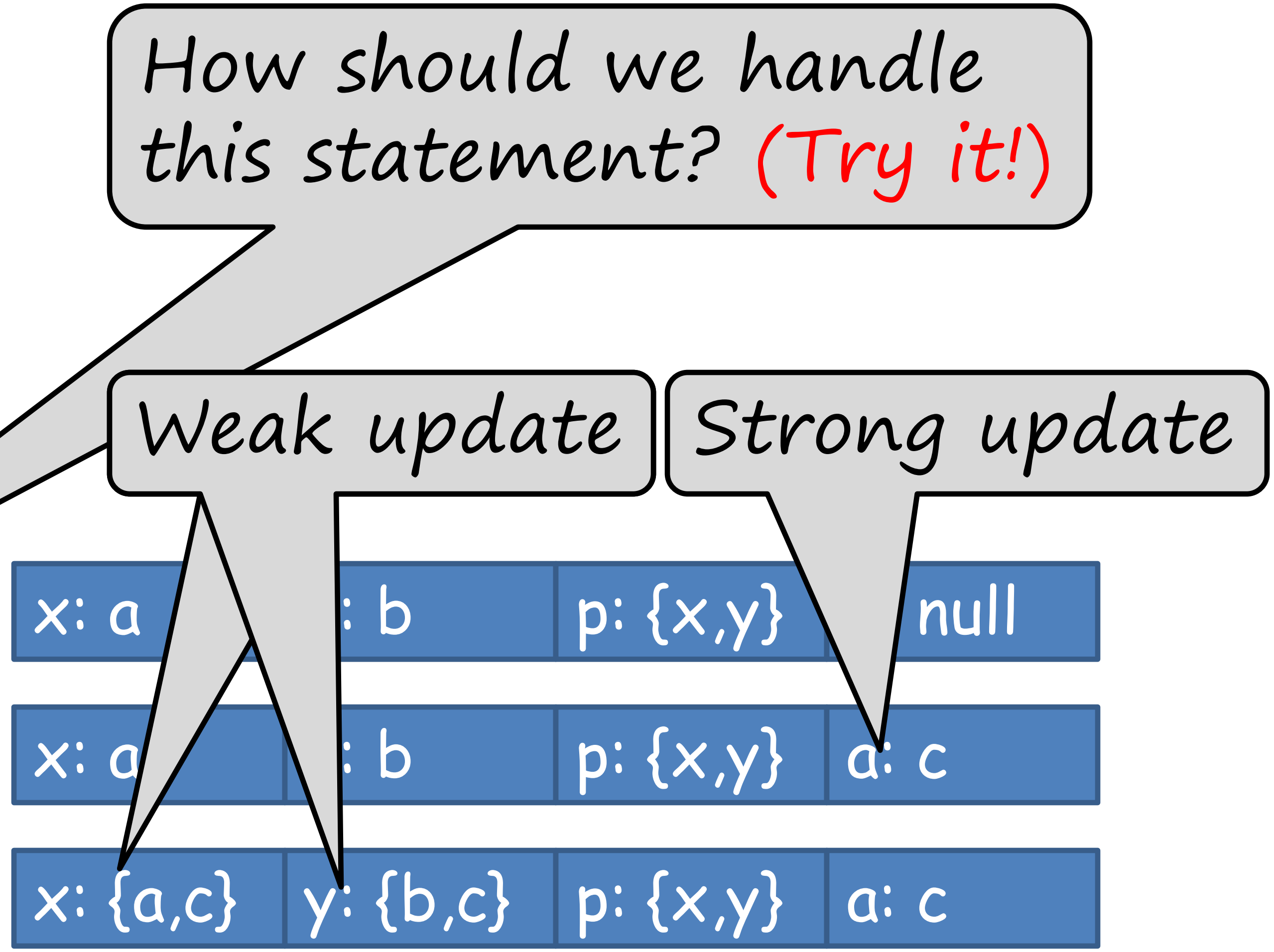
```
p = &x;
q = &y;
if (?) {
   q = p;
}
x = &a;
y = &b;
z = *q;
```

p	q	x	y	z

Algorithm A (may points-to analysis)
A Simple Example

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

*x = &c;
*p = &c;
```



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 IdealMayPT (u) to be a function

  \p. {x | p points-to x in some state at u in some run }

- Algorithm should compute a function MayPT(u) that over-approximates above function

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 Var
  - a directed graph (V,E,entry) with a distinguished entry vertex, with every edge labelled by a primitive statement
- A primitive statement is of the form
  - x = null
  - x = y
  - x = *y
  - x = &y;
  - *x = y
  - skip

(where x and y are variables in Var)

Omitted (for now)

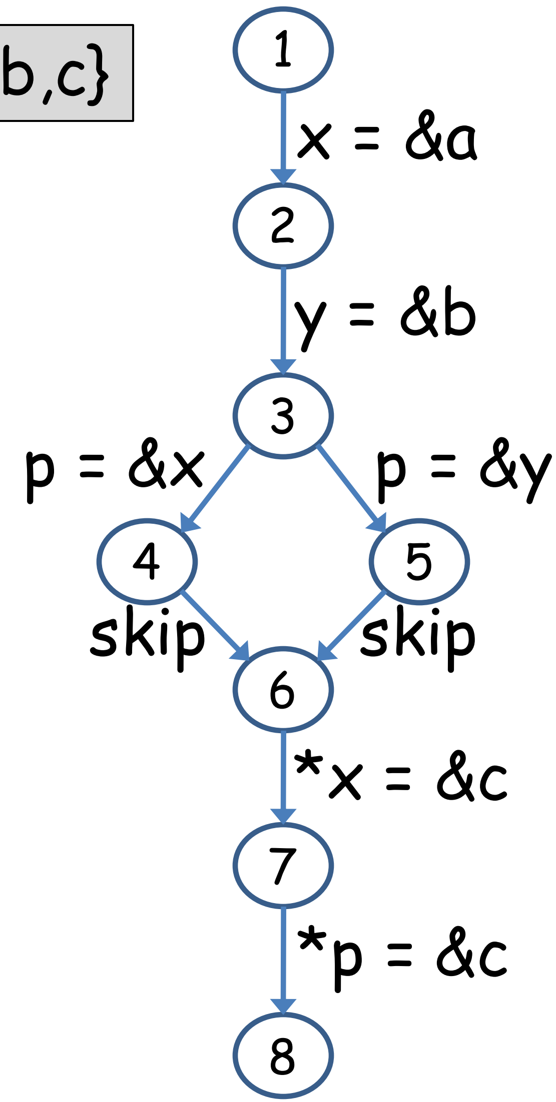
- Dynamic memory allocation
- Pointer arithmetic
- Structures and fields
- Procedures

Example Program

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

*x = &c;
*p = &c;
```

Vars = {x,y,p,a,b,c}

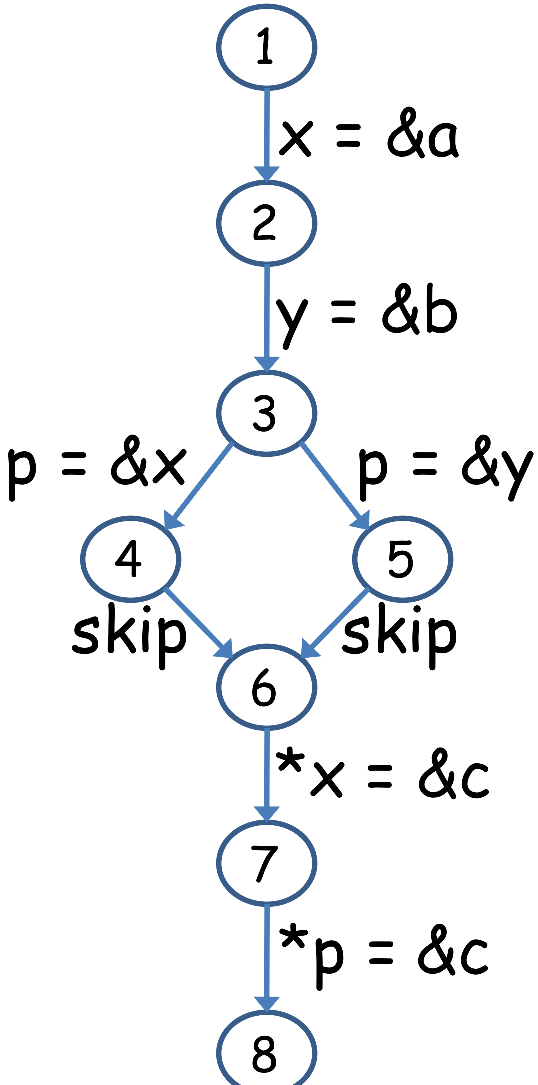


Programming Language: Operational Semantics

- Operational semantics == an interpreter (defined mathematically)
- State
  - DataState ::= Var -> (Var U {null})
- Initial-state:
  - \x. null

Example States

Vars = {x,y,p,a,b,c}



Initial data-state

x: N, y:N, p:N, a:N, b:N, c:N

Initial program-state

<1, x: N, y:N, p:N, a:N, b:N, c:N >

Next program-state

<2, x: a, y:N, p:N, a:N, b:N, c:N >

Programming Language: Operational Semantics

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

- CS[x = null] s = $\{s[x \rightarrow null]\}$
- CS[x = &y] s = $\{s[x \rightarrow y]\}$
- CS[x = y] s = $\{s[x \rightarrow s(y)]\}$
- CS[x = *y] s = …

…

…

- CS[*x = y] s = …

…

= …

Programming Language: Operational Semantics

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

- CS[x = null] s = {s[x → null]}
- CS[x = &y] s = {s[x → y]}
- CS[x = y] s = {s[x → s(y)]}
- CS[x = *y] s = {s[x → s(s(y))]},
  if s(y) is not null
  = {}, otherwise
- CS[*x = y] s = …

…

…

Programming Language: Operational Semantics

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

- CS[x = null] s = {s[x → null]}
- CS[x = &y] s = {s[x → y]}
- CS[x = y] s = {s[x → s(y)]}
- CS[x = *y] s = {s[x → s(s(y))]}, if s(y) is not null
  = {}, otherwise
- CS[*x = y] s = {s[s(x) → s(y)]}, if s(x) is not null
  = {}, otherwise

Programming Language: Operational Semantics

- Let u denote a vertex in the CFG
- Define RS(u) = { s | s is a DataState that can arise at point u in some execution }
  - It is the collecting semantics at u

May-Point-To Analysis: Problem statement

Compute MayPT: V -> $2^{Var'}$ such that for every vertex u MayPT(u) $\supseteq$ IdealMayPT(u), where Var' = Var U {null}.

Given two functions f and g, we say f $\supseteq$ g, iff forall x
f(x) $\supseteq$ g(x)

May-Point-To Algorithms

Compute MayPT: V -> $2^{Vars'}$ such that
MayPT(u) ⊇ 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 ::=

    $(Var \to (2^{Var'} - \{\})) \cup \{bot\}$
  - $f_1 \cup f_2 = \backslash x.\ (f_1(x) \cup f_2(x))$
- Define initial abstract-value
  - InitialAbsState = \x. {null}
- Define transformers for primitive statements
    - AS[stmt] : AbsDataState -> AbsDataState

Algorithm A: A Formal Definition
The "Data Flow Analysis" Recipe

- Apply Kildall's algorithm, using AbsDataState lattice, and AS transfer functions.

Algorithm A: The Transformers

- Abstract transformers for primitive statements
  - AS[stmt] : AbsDataState -> AbsDataState
- AS[x = y] s = $s[x \rightarrow s(y)]$
- AS[x = null] s = $s[x \rightarrow \{null\}]$
- AS[x = &y] s = $s[x \rightarrow \{y\}]$
- AS[x = *y] s = $s[x \rightarrow s^*(s(y) - \{null\})]$, if s(y) is not = {null}
  = bot, otherwise

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

Algorithm A

- AS[*x = y] s =

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

- After fix-point solution is obtained, AbsDataState(u) is emitted as MayPT(u), for each program point 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 $\Upsilon(S) = S$,
if S is a singleton set
= Var', if S = {}

Must points-to analysis algorithm

- AS transfer functions same as in Algorithm A for $x = y$, $x = null$, and $x = \&y$
- AS[$x = *y$] s

$$
\begin{aligned}
&= bot, && \text{if } s(y) = \{null\} \\
&= s[x \rightarrow \{\}], && \text{if } s(y) = \{\} \\
&= s[x \rightarrow s(z)], && \text{if } s(y) = \{z\}
\end{aligned}
$$

Must points-to analysis algorithm

- AS[*x = y] s = bot,

  if s(x) = {null}

  = s[z → s(y)]

  if s(x) = {z}

  = \v. {},

  otherwise

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