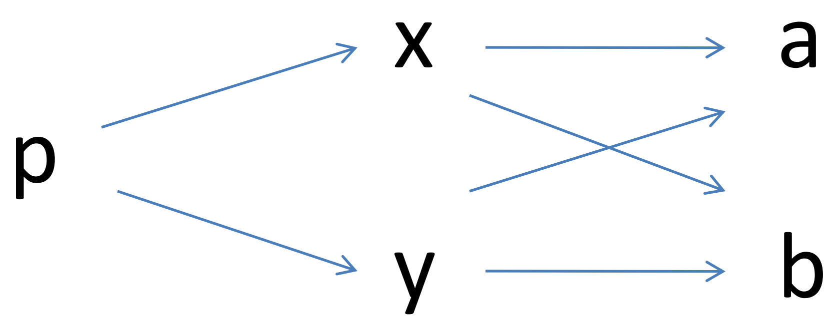
Points-to Analysis in Almost Linear Time

Bjarne Steensgard

Constraint-based Analysis

- Idea: generate constraints and solve them later

```
x = &a;
y = &b;
p = &x;
p = &y;
```



Inclusion-based Analysis

$x = y$

$pointsTo(x) \geq pointsTo(y)$

What is the major drawback of this approach?

$O(n^3)$

How can we do this faster?

- Use equality-based analysis. Why?

Equality-based Analysis

x = y

pointsTo(x) = pointsTo(y)

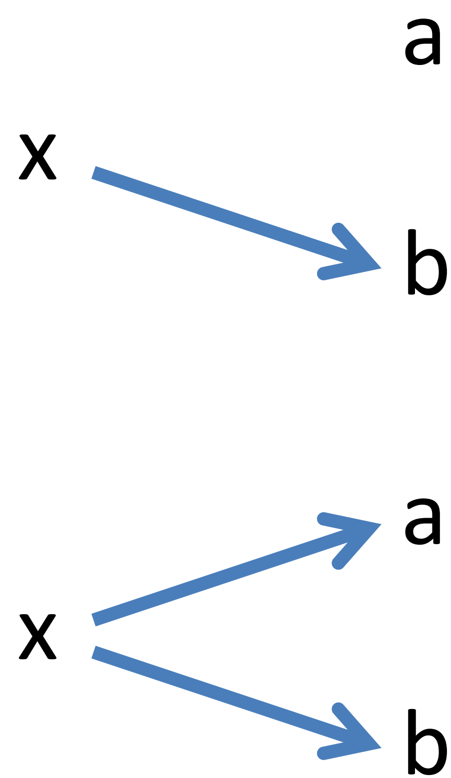
Why is this faster?

What are the tradeoffs?

What should x point to?

x = a

x = b



Imprecise, but fast – really?

- How to do equality-based, flow-insensitive analysis in one pass?
- Use type inference with points-to sets as types
  - For every variable X, let X's type α_x = pointsTo(X)
  - The set $\{\alpha_x\}$ – the goal of the analysis – is found using unification-based type inference
- How is this analysis equality-based?

Type system for points-to inference

3 kinds of types:

- Value types – (pointer, function) tuples
  - $\alpha ::= \tau \times \lambda$
- pointer/address types:
  - $\tau ::=$ **ref**(α) | $\perp$ *(null, or actual value / not pointer)*
- function signatures:
  - $\lambda ::= (\alpha_1,\ldots\alpha_n) \rightarrow (\alpha_{n+1},\ldots\alpha_{n+m}) \mid \perp$

Type inequality / compatibility: $\leq$

- For atomic types α_1 and α_2:
  - $\alpha_1 \leq \alpha_2$ iff $\alpha_1 = \alpha_2$ or α_1 is $\bot$
- For composite types, component types must be compatible recursively

Type rules induce points-to constraints

Example: assignment "x = y", under type environment A:

$A \vdash x : \mathbf{ref}(\alpha_1)$

$A \vdash y : \mathbf{ref}(\alpha_2)$

$\alpha_2 \leq \alpha_1$

$\Rightarrow A \vdash \text{well-typed}(x = y)$

Why does this only make sense for equality-based analysis?

Other type rules

- Simple language with fairly obvious typing rules
  - Assignment of one variable to another (plus dereference on either side, address-of on right)
  - Using built-in operators
  - malloc()
  - Function definition and call

Algorithm: Infer Types

- Consider the following program:

```
x = &a;
y = &b;
p = &x;
p = &y;
```

Algorithm: Initialize Types

```
x = &a;
y = &b;
p = &x;
p = &y;
```

x : t1

y : t2

a : t3

b : t4

p : t5

Algorithm: Initial Constraints

$x : t1$

$y : t2$

$a : t3$

$b : t4$

$p : t5$

x = &a;	$t1 = \textbf{ref}(t3 \times \perp)$
y = &b;	$t2 = \textbf{ref}(t4 \times \perp)$
p = &x;	$t5 = \textbf{ref}(t1 \times \perp)$
p = &y;	$t5 = \textbf{ref}(t2 \times \perp)$

Algorithm: Joining

```
x = &a;
y = &b;
p = &x;
p = &y;
```

$x : t1$

$y : t1$

$a : t3$

$b : t4$

$p : t5$

$t1 = \mathbf{ref}(t3 \times \bot)$

$t1 = \mathbf{ref}(t4 \times \bot)$

$t5 = \mathbf{ref}(t1 \times \bot)$

Algorithm: Joining

```
x = &a;
y = &b;
p = &x;
p = &y;
```

x : t1

y : t1

a : t3

b : t3

p : t5

$t1 = \mathbf{ref}(t3 \times \perp)$

$t5 = \mathbf{ref}(t1 \times \perp)$

Algorithm: End

$t1 = ref(t3 \times \perp)$

$t5 = ref(t1 \times \perp)$

t5 -> t1 -> t3

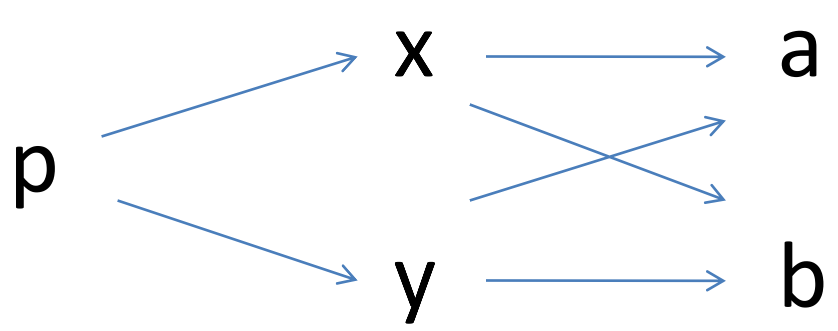
$x : t1$

$y : t1$

$a : t3$

$b : t3$

$p : t5$



Algorithm

- What about values that are never a pointer?

- Conditional join
  - If left-hand side has type _ , add right-hand side variable to left-hand set
  - If left-hand side has type other than _ , do real join

Data Structures

- Fast union-find

Time Complexity

- What is the time complexity of this algorithm?
- Cost of traversing program statements + cost of creating type variable data structures + cost of joins
- First two are proportional to size of input program, N
- Joins: $O(N\alpha(N,N))$, where α is an inverse Ackermann's function (grows slowly)

Results

- Can analyze 100,000 line programs (up from about 10,000 lines)

- Did not find anything interesting in the code

- How effective is this method?