Bjarne Steensgaard

Points-to Analysis in Almost Linear Time

Presented by **Abdul Dakkak**

Overview

- Definitions
- Algorithm Overview
- Language Overview
- Algorithm Details
- Results
- Conclusion

Definitions

Definitions

- A variable p *points-to* a value v if p's value may contain the address of v
- Two values p and q alias if they may point-to the same variable (if $\text{pts}(p) \cap \text{pts}(q) \neq \{\}$)
- The *points-to set* for p contains the set of variables which p may point-to ($v \in \text{pts}(p)$ iff p may point-to v)

Definitions

- An analysis is *flow sensitive* if it performs the analysis at each program point and follows the instruction flow (keeps track of branches, definition kills, …)
- An analysis is *context sensitive* if keeps flow along different call paths separate (considers calling context when performing the analysis)
- An analysis if *field sensitive* if it distinguishes between elements in a field (**struct** in C)

Unification

- A unification algorithm finds a set of replacement rules that make two terms equal
    - e.x. if f(g) = f(a) then g → a is a substitution that makes the two terms equal
- For Steensgard, unification means union

Pointer Analysis Uses

- Pointer analysis facilitates compiler optimizations
- Useful only when you know that two variables *must not* alias

CSE: If **p* aliases *a* or *b* then we cannot replace the recomputation of *a*+*b* with *r*.

```
add $r, $a, $b
sw  $r, ($p)
add $g, $a, $b
```

Constant Propagation: If **p* aliases *x* then we cannot propagate the value of *x* via *r*.

```
lw $r, $x
sw $y, ($p)
lw $v, $x
```

Algorithm Overview

Steensgaard's Algorithm

- Flow insensitive, interprocedural, context insensitive unification based pointer analysis
- Performs pointer analysis in near linear time
- Traverse the instructions from top to bottom in a single pass

Algorithm Overview

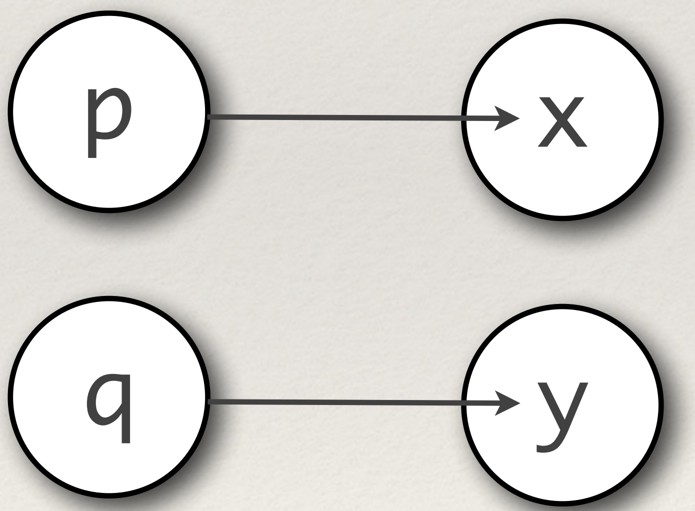
- Perform the analysis instruction by instruction
- If `p = &x` then p points-to x
- If `p = q` , then $pts(\texttt{p}) = pts(\texttt{q}) = pts_{old}(\texttt{p}) \cup pts_{old}(\texttt{q})$

```
p = &x
q = &y
p = q
s = &p
```

Algorithm Overview

- Perform the analysis instruction by instruction
- If `p = &x` then p points-to x
- If `p = q`, then $pts(\mathsf{p}) = pts(\mathsf{q}) = pts_{old}(\mathsf{p}) \cup pts_{old}(\mathsf{q})$

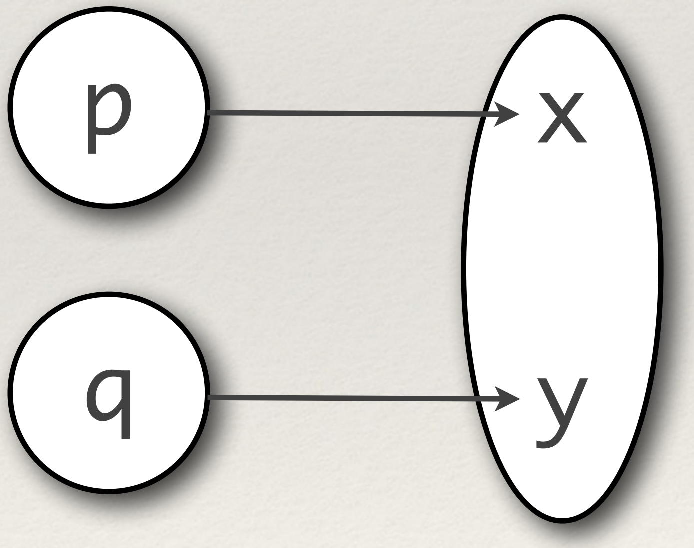
```
p = &x
q = &y
p = q
s = &p
```



Algorithm Overview

- Perform the analysis instruction by instruction
- If `p = &x` then p points-to x
- If `p = q`, then $pts(\mathtt{p}) = pts(\mathtt{q}) = pts_{old}(\mathtt{p}) \cup pts_{old}(\mathtt{q})$

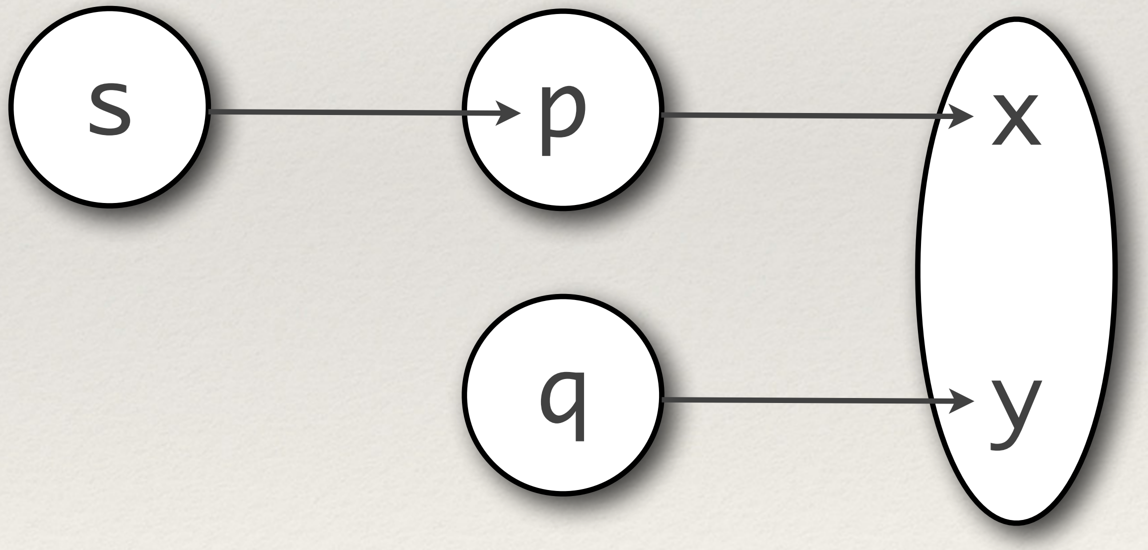
```
p = &x
q = &y
p = q
s = &p
```



Algorithm Overview

- Perform the analysis instruction by instruction
- If `p = &x` then p points-to x
- If `p = q`, then $pts(\mathtt{p}) = pts(\mathtt{q}) = pts_{old}(\mathtt{p}) \cup pts_{old}(\mathtt{q})$

```
p = &x
q = &y
p = q
s = &p
```



Steensgaard's Language

Steensgaard's Language

- Represent C programs as values and operations on addresses of these values
- Function can have multiple return values
- Distinction between function calls and function operations on addresses

```
S ::= x =  y
    | x = &y
    | x = *y
    |*x =  y
    | x =  op(y...)
    | x =  allocate(y)
    | x =  fun(a...) -> (r...) S*
    | x... = p(a...)
```

Translating into Steensgaard's Language

```
S ::= x =  y
    | x = &y
    | x = *y
    |*x =  y
    | x =  op(y...)
    | x =  allocate(y)
    | x =  fun(a...) -> (r...) S*
    | x... = p(a...)
```

Variables within a function scope have unique names

```
int addNumbers(int a, int b) {
   int res = a + b;
   return res;
}
```

```
addNumbers = fun(a0, b0) -> (res0)
   res0 = a0 + b0;
```

Translating into Steensgaard's Language

```
S ::= x =  y
    | x = &y
    | x = *y
    |*x =  y
    | x =  op(y...)
    | x =  allocate(y)
    | x =  fun(a...) -> (r...) S*
    | x... = p(a...)
```

Structures and their accessors are collapsed

```
struct Point_t {
     int x, y;
};
...
struct Point_t pt;
pt.x = 3;
pt.y = 4;
```

```
pt = 3
pt = 4
```

Translating into Steensgaard's Language

```
S ::= x =  y
    | x = &y
    | x = *y
    |*x =  y
    | x =  op(y...)
    | x =  allocate(y)
    | x =  fun(a...) -> (r...) S*
    | x... = p(a...)
```

Pointer operations are normalized by introducing temporary variables or function calls

```
p = **a;
q = p + 1;
```

```
p0 = *a
p  = *p0
q  = add(p, 1)
```

Algorithm Details

Steensgaard's Algorithm

- Flow insensitive, interprocedural, context insensitive unification based pointer analysis
- Performs pointer analysis in near linear time
- Traverse the instructions from top to bottom in a single pass

Modeling Values

- Model values as pointers to locations or functions
  - **Value**: $\boldsymbol{\alpha} = \boldsymbol{\tau} \times \boldsymbol{\lambda}$
  - **Pointer/Address data type**: $\boldsymbol{\tau} = \mathrm{ref}(\boldsymbol{\alpha}) \mid \perp$
  - **Pointer to function**: $\lambda = \mathrm{lam}(\alpha_1, \ldots)(\alpha_k, \ldots) \mid \perp$
- Each location in the program either contains a function or data
- Keeps space usage to $O(n)$ using this representation
- Impose partial order on memory locations

Partial Ordering

- We define a partial Ordering operator $\trianglelefteq$ such that
  - $(t_1 \trianglelefteq t_2) \Leftrightarrow (t_1 = \perp) \vee (t_1 = t_2)$
  - $(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4)$
- Type type of each type variable is initially assumed to be $\mathrm{ref}(\perp\times\perp)$

Why Partial Ordering Matters

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \qquad A \vdash \mathsf{y} : \mathbf{ref}(\alpha)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

```
a = 2
x = a
y = a
```

*Would incorrectly alias **x** and **y***

Typing Rules

```
p = &x
q = &y
p = q
s = &p
```

```
x : t0
t : t1
```

- Values are initialized to ref(⊥×⊥)

Typing Rules

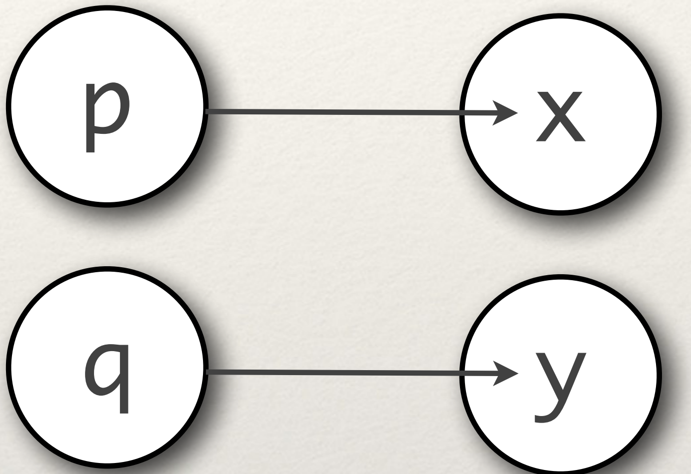
```
p = &x
q = &y
p = q
s = &p
```

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \qquad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

```
x : t0 = ref(⊥×⊥)
t : t1 = ref(⊥×⊥)
p : t2 = ref(t0 × ⊥)
```

Typing Rules

```
p = &x
q = &y
p = q
s = &p
```

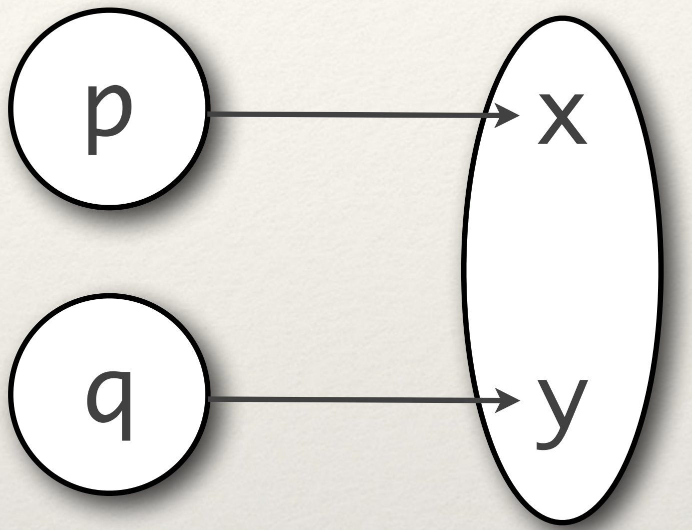


$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \quad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

```
x : t0 = ref(⊥×⊥)
t : t1 = ref(⊥×⊥)
p : t2 = ref(t0 × ⊥)
q : t3 = ref(t1 × ⊥)
```

Typing Rules

```
p = &x
q = &y
p = q
s = &p
```

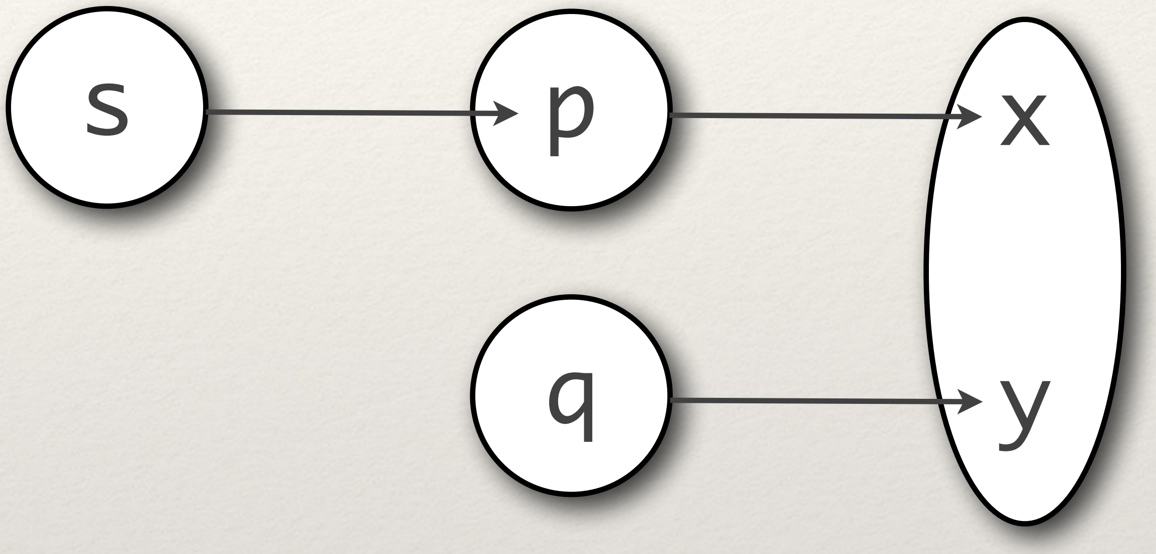


$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

x : t0 = ref(⊥×⊥)
t : t0 = ref(⊥×⊥)
p : t2 = ref(t0 × ⊥)
q : t3 = ref(t0 × ⊥)

Typing Rules

```
p = &x
q = &y
p = q
s = &p
```

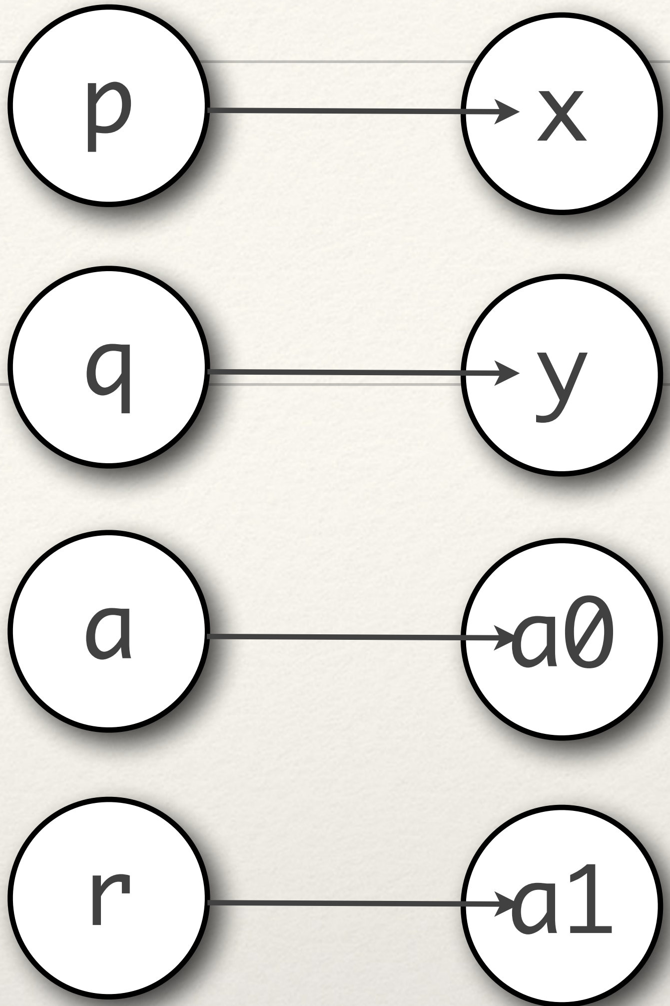


$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \qquad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

```
x : t0 = ref(⊥×⊥)
t : t0 = ref(⊥×⊥)
p : t2 = ref(t0 × ⊥)
q : t3 = ref(t0 × ⊥)
s : t4 = ref(t2 × ⊥)
```

Typing Functions

```
p = &x
q = &y
id = fun(a) -> (r)
     r = a
x = id(p)
y = id(q)
```



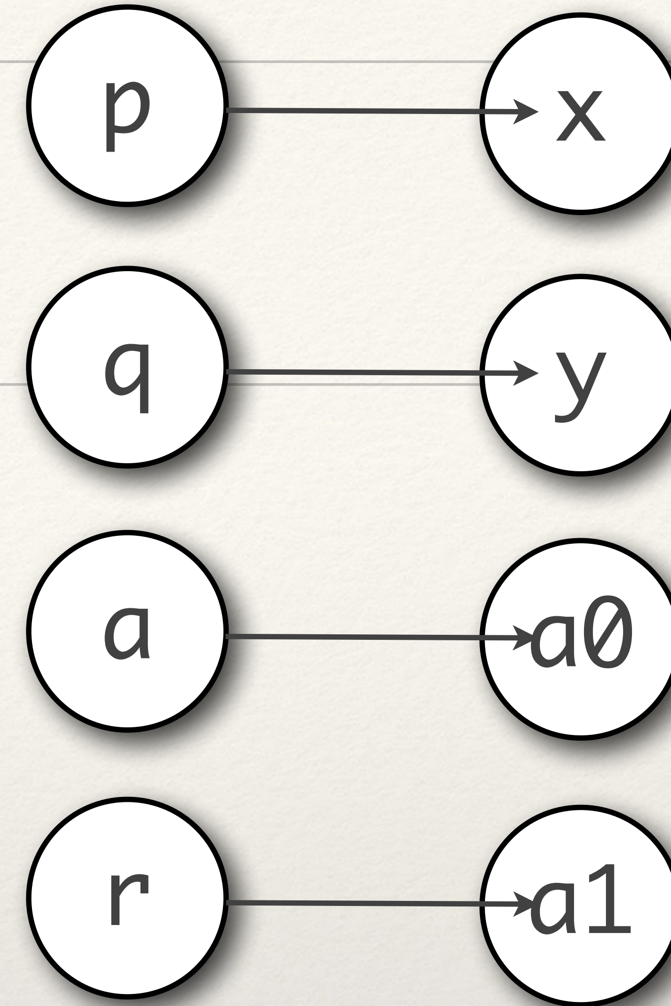
x: t0 = ref(⊥×⊥)
y: t1 = ref(⊥×⊥)
p: t2 = ref(t0 ×⊥)
q: t3 = ref(t1 ×⊥)
a0: t4 = ref(⊥×⊥)
a1: t5 = ref(⊥×⊥)
id: t6 = ref(⊥×lam(t4)(t5))
a : t7 = ref(t4)
r : t8 = ref(t5)

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{f}_i : \mathbf{ref}(\alpha_i) \\ A \vdash \mathsf{r}_j : \mathbf{ref}(\alpha_{n+j}) \\ \forall s \in S^* : A \vdash \mathit{welltyped}(s) \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^*)}$$

Typing Functions

```
p = &x
q = &y
id = fun(a) -> (r)
    r = a
x = id(p)
y = id(q)
```

x: t0 = ref(⊥×⊥)
y: t1 = ref(⊥×⊥)
p: t2 = ref(t0 ×⊥)
q: t3 = ref(t1 ×⊥)
a0: t4 = ref(⊥×⊥)
a1: t5 = ref(⊥×⊥)
id: t6 = ref(⊥×lam(t4)(t5))
a : t7 = ref(t4)
r : t8 = ref(t5)



$$\frac{\begin{array}{c} A \vdash \mathsf{x}_j : \mathbf{ref}(\alpha'_{n+j}) \\ A \vdash \mathsf{p} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha'_i) \\ \forall i \in [1 \ldots n] : \alpha'_i \trianglelefteq \alpha_i \\ \forall j \in [1 \ldots m] : \alpha_{n+j} \trianglelefteq \alpha'_{n+j} \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

Steensgaard Unifies Function Arguments

- If $t_1 \trianglelefteq C$ and $t_2 \trianglelefteq C$ then $t_1 = t_2$
  - Recall $(t \trianglelefteq C) \Leftrightarrow (t = \perp) \vee (t = C)$
- This means that if we call a function with different arguments, then we will unify the arguments
- When run on real programs most pointers will alias each other.

```
memset(s, 0, sz);
memset(k, 0, sz);
strcmp(p, q);
free(s);
```

Results

Unoptimized Results

18 classes were empty.

67 classes contained 1 variable.

One equivalence class contained 120 variables.

# of vars.	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6 . . .	30	31	32	33 . . .	44	45 . . .	52 . . .	74 . . .	78 . . .	83 . . .	113 . . .	120 . . .	285 . . .	613 . . .	624
landi:allroots	18	67																																								
landi:assembler	157	446	3		3	1					1																				1								1			
landi:loader	90	211	1	1			1	1						1																												
landi:compiler	116	166																					1										1									
landi:simulator	232	464	3	1	2	1			2				2		1								1	2										2			1					
landi:lex315	52	91																						1																		
landi:football	214	570	15	14	1					1																1																
austin:anagram	54	65					1		1																																	
austin:backprop	43	69																																								
austin:bc	297	551	2													2			2																	1						
austin:ft	61	150																																								
austin:ks	68	158	2			1																																				
austin:yacr2	260	474	3	1																																						
spec:compress	88	113	1																																							
spec:eqntott	228	437	2				1	1	1																																	
spec:espresso	1155	2556	14	3	2	1	2	1								1	1						1							1		1			1							
spec:li	449	877	1	2	1		2																					1														1
spec:sc	557	1000	26	4	3			1	2						1										1													1		1		
spec:alvinn	43	73																																								
spec:ear	192	532	3		2			1																			1															
LambdaMOO	1369	2580	9	2	3		1	2																	1								1								1	

Conclusion

Author's Conclusions

- First algorithm to scale to 100kLOC (previous methods only scaled to 10kLOC)
- Uses very little memory and performs the analysis in a reasonable amount of time

Conclusion

- First algorithm to scale beyond 10kLOC
- Previous state of the art pointer analysis was O(n^3)
- Can be used as a first pass for pointer analysis, more precise algorithms can be used on the refined subset
- SSA provides some flow sensitivity for pointer analysis
- Not used in practice because it unifies function arguments

Questions

Thank You

Backup Slides

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \\ A \vdash \mathsf{y} : \tau \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\mathbf{ref}(\alpha_2) \times \_) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = *\mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \\ A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha_i) \\ \forall i \in [1 \ldots n] : \alpha_i \trianglelefteq \alpha \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ \ldots \ \mathsf{y}_n))}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\_) \times \_)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{allocate}(\mathsf{y}))}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\alpha_1) \times \_) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(*\mathsf{x} = \mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{f}_i : \mathbf{ref}(\alpha_i) \\ A \vdash \mathsf{r}_j : \mathbf{ref}(\alpha_{n+j}) \\ \forall s \in S^* : A \vdash \mathit{welltyped}(s) \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m) \ S^*)}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x}_j : \mathbf{ref}(\alpha'_{n+j}) \\ A \vdash \mathsf{p} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha'_i) \\ \forall i \in [1 \ldots n] : \alpha'_i \trianglelefteq \alpha_i \\ \forall j \in [1 \ldots m] : \alpha_{n+j} \trianglelefteq \alpha'_{n+j} \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

Type Rules

Implementation

```
x = y:
  let ref(τ1 × λ1) = type(ecr(x))
      ref(τ2 × λ2) = type(ecr(y)) in
    if τ1 ≠ τ2 then cjoin(τ1, τ2)
    if λ1 ≠ λ2 then cjoin(λ1, λ2)
x = &y:
  let ref(τ1 × _) = type(ecr(x))
      τ2 = ecr(y) in
    if τ1 ≠ τ2 then join(τ1, τ2)
x = *y:
  let ref(τ1 × λ1) = type(ecr(x))
      ref(τ2 × _) = type(ecr(y)) in
    if type(τ2) = ⊥ then
      settype(τ2, ref(τ1 × λ1))
    else
      let ref(τ3 × λ3) = type(τ2) in
        if τ1 ≠ τ3 then cjoin(τ1, τ3)
        if λ1 ≠ λ3 then cjoin(λ1, λ3)
x = op(y1 ... yn):
  for i ∈ [1...n] do
    let ref(τ1 × λ1) = type(ecr(x))
        ref(τ2 × λ2) = type(ecr(yi)) in
      if τ1 ≠ τ2 then cjoin(τ1, τ2)
      if λ1 ≠ λ2 then cjoin(λ1, λ2)
x = allocate(y):
  let ref(τ × _) = type(ecr(x)) in
    if type(τ) = ⊥ then
      let [e1, e2] = MakeECR(2) in
        settype(τ, ref(e1 × e2))
*x = y:
  let ref(τ1 × _) = type(ecr(x))
      ref(τ2 × λ2) = type(ecr(y))
    if type(τ1) = ⊥ then
      settype(τ1, ref(τ2 × λ2))
    else
      let ref(τ3 × λ3) = type(τ1) in
        if τ2 ≠ τ3 then cjoin(τ3, τ2)
        if λ2 ≠ λ3 then cjoin(λ3, λ2)
x = fun(f1 ... fn)→(r1 ... rm) S*:
  let ref(_ × λ) = type(ecr(x))
    if type(λ) = ⊥ then
      settype(λ, lam(α1 ... αn)(αn+1 ... αn+m))
      where
        ref(αi) = type(ecr(fi)), for i ≤ n
        ref(αi) = type(ecr(ri−n)), for i > n
    else
      let lam(α1 ... αn)(αn+1 ... αn+m) = type(λ) in
        for i ∈ [1...n] do
          let τ1 × λ1 = αi
              ref(τ2 × λ2) = type(ecr(fi)) in
            if τ1 ≠ τ2 then join(τ2, τ1)
            if λ1 ≠ λ2 then join(λ2, λ1)
        for i ∈ [1...m] do
          let τ1 × λ1 = αn+i
              ref(τ2 × λ2) = type(ecr(ri)) in
            if τ1 ≠ τ2 then join(τ1, τ2)
            if λ1 ≠ λ2 then join(λ1, λ2)
x1 ... xm = p(y1 ... yn):
  let ref(_ × λ) = type(ecr(p)) in
    if type(λ) = ⊥ then
      settype(λ, lam(α1 ... αn)(αn+1 ... αn+m))
      where
        αi = τi × λi
        [τi, λi] = MakeECR(2)
    let lam(α1 ... αn)(αn+1 ... αn+m) = type(λ) in
      for i ∈ [1...n] do
        let τ1 × λ1 = αi
            ref(τ2 × λ2) = type(ecr(yi)) in
          if τ1 ≠ τ2 then cjoin(τ1, τ2)
          if λ1 ≠ λ2 then cjoin(λ1, λ2)
      for i ∈ [1...m] do
        let τ1 × λ1 = αn+i
            ref(τ2 × λ2) = type(ecr(xi)) in
          if τ1 ≠ τ2 then cjoin(τ2, τ1)
          if λ1 ≠ λ2 then cjoin(λ2, λ1)
```

$$
\begin{array}{l}
\textbf{settype}(e, t)\text{:} \\
\quad \textbf{type}(e) \leftarrow t \\
\quad \text{for } x \in \textbf{pending}(e) \text{ do } \textbf{join}(e, x) \\
\\
\textbf{cjoin}(e_1, e_2)\text{:} \\
\quad \text{if } \textbf{type}(e_2) = \perp \text{ then} \\
\quad\quad \textbf{pending}(e_2) \leftarrow \{e_1\} \cup \textbf{pending}(e_2) \\
\quad \text{else} \\
\quad\quad \textbf{join}(e_1, e_2) \\
\\
\textbf{unify}(\textbf{ref}(\tau_1 \times \lambda_1), \textbf{ref}(\tau_2 \times \lambda_2))\text{:} \\
\quad \text{if } \tau_1 \neq \tau_2 \text{ then } \textbf{join}(\tau_1, \tau_2) \\
\quad \text{if } \lambda_1 \neq \lambda_2 \text{ then } \textbf{join}(\lambda_1, \lambda_2) \\
\\
\textbf{unify}(\textbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}), \\
\qquad\quad \textbf{lam}(\alpha'_1 \ldots \alpha'_n)(\alpha'_{n+1} \ldots \alpha'_{n+m})) \\
\quad \text{for } i \in [1 \ldots (n+m)] \text{ do} \\
\quad \text{let } \tau_1 \times \lambda_1 = \alpha_i \\
\quad\quad\; \tau_2 \times \lambda_2 = \alpha'_i \text{ in} \\
\quad\quad \text{if } \tau_1 \neq \tau_2 \text{ then } \textbf{join}(\tau_1, \tau_2) \\
\quad\quad \text{if } \lambda_1 \neq \lambda_2 \text{ then } \textbf{join}(\lambda_1, \lambda_2)
\end{array}
$$

$$
\begin{array}{l}
\textbf{join}(e_1, e_2)\text{:} \\
\quad \text{let } t_1 = \textbf{type}(e_1) \\
\quad\quad\;\; t_2 = \textbf{type}(e_2) \\
\quad\quad\;\; e = \text{ecr-union}(e_1, e_2) \text{ in} \\
\quad\quad \text{if } t_1 = \perp \text{ then} \\
\quad\quad\quad \textbf{type}(e) \leftarrow t_2 \\
\quad\quad\quad \text{if } t_2 = \perp \text{ then} \\
\quad\quad\quad\quad \textbf{pending}(e) \leftarrow \textbf{pending}(e_1) \cup \\
\quad\quad\quad\quad\qquad\qquad\quad \textbf{pending}(e_2) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \text{for } x \in \textbf{pending}(e_1) \text{ do } \textbf{join}(e, x) \\
\quad\quad \text{else} \\
\quad\quad\quad \textbf{type}(e) \leftarrow t_1 \\
\quad\quad\quad \text{if } t_2 = \perp \text{ then} \\
\quad\quad\quad\quad \text{for } x \in \textbf{pending}(e_2) \text{ do } \textbf{join}(e, x) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \textbf{unify}(t_1, t_2)
\end{array}
$$

Unification

Flow Sensitive vs Insensitive Analysis

Flow Sensitive: Pointer analysis follows the CFG. Computes the points-to set at all program points.

$$\begin{array}{ll} & y \\ p \nearrow & \\ & x \end{array}$$

```
x = 1;
y = 2;
p = &x;
p = &y;
```

Flow Insensitive: Pointer analysis ignores the order of statements in the program.

$$\begin{array}{ll} & y \\ p \begin{array}{l}\nearrow \\ \searrow\end{array} & \\ & x \end{array}$$

SSA Remedies Some Flow Insensitivity

Flow Sensitive: Pointer analysis follows the CFG. Computes the points-to set at all program points.

$$p1 \longrightarrow y$$

$$p0 \longrightarrow x$$

```
x = 1;
y = 2;
p0 = &x;
p1 = &y;
```

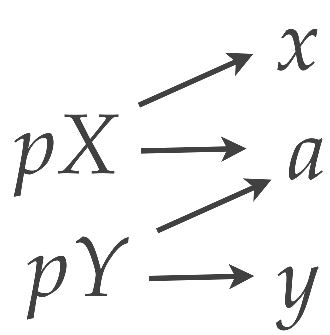
Flow Insensitive: Pointer analysis ignores the order of statements in the program.

$$p1 \longrightarrow y$$

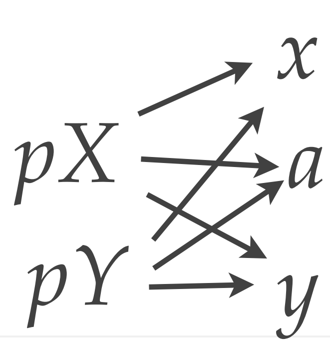
$$p0 \longrightarrow x$$

Context Sensitive vs Insensitive Analysis

Context Sensitive: Considers calling context when performing *points-to* analysis.



Context Insensitive: Produces spurious aliases.




```
void * id(void * a) {
    return a;
}
void fa() {
    int x = 1;
    void * pX = &x;
    pX = id(pX);
}
void fb() {
    int y = 1;
    void * pY = &y;
    pY = id(pY);
}
```