Bjarne Steensgaard

Points-to Analysis in Almost Linear Time

Presented by **Abdul Dakkak**

Overview

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

Algorithm Overview

Pointer Analysis Uses

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

Redundant Store Elimination: If *$R0 and *$R1 alias then we cannot remove the second store to the address stored in $R0

```
sw 1, ($R0)  //*$R0=1
sw 2, ($R1)  //*$R1=2
sw 1, ($R0)  //*$R0=1
```

Constant Propagation: If *R1 and *$R2 aliases then we cannot propagate the value of *$R1 via *$R0

```
lw $R0, ($R1)  //$R0=*$R1
sw 1,   ($R2)  //*$R2=1
lw $R3, ($R1)  //$R3=*$R1
```

Steensgaard's Algorithm

- Previous work, such as address taken, is too imprecise
- Previous work, such as Andersen's, is more precise, but has a large complexity to be used for real programs
- Find a middle ground that is more precise than address taken while being faster than Andersen

Algorithm Overview

- Build a graph where each object is a node and an edge represents the points-to relation
    - Add nodes x and p
    - Create an edge $p \rightarrow x$

Program

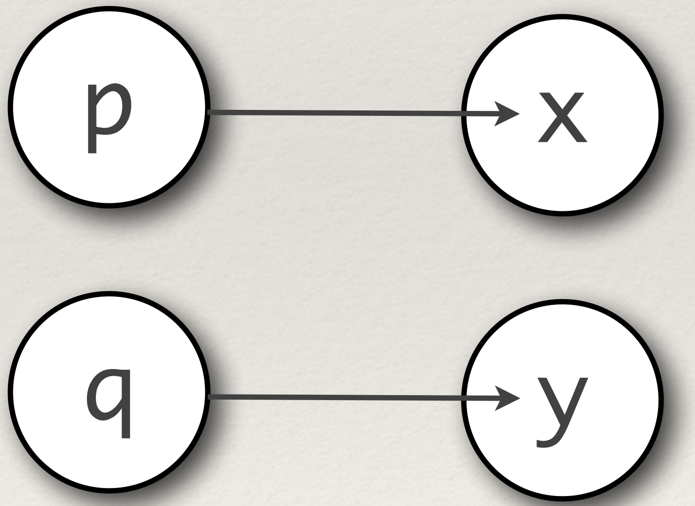
```
p = &x
q = &y
p = q
s = &p
```

Points-to Graph

Algorithm Overview

- Build a graph where each object is a node and an edge represents the points-to relation
  - Add nodes *y* and *q*
  - Create an edge $q \rightarrow y$

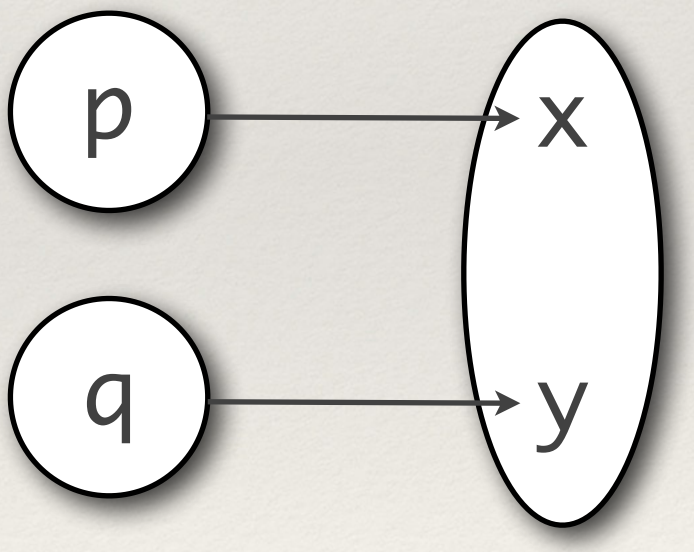
```
p = &x
q = &y
p = q
s = &p
```



Algorithm Overview

- Collapse the nodes pointed to by p and q
  - Each nodes is now a collection of objects
  - Each node my point-to at most one node

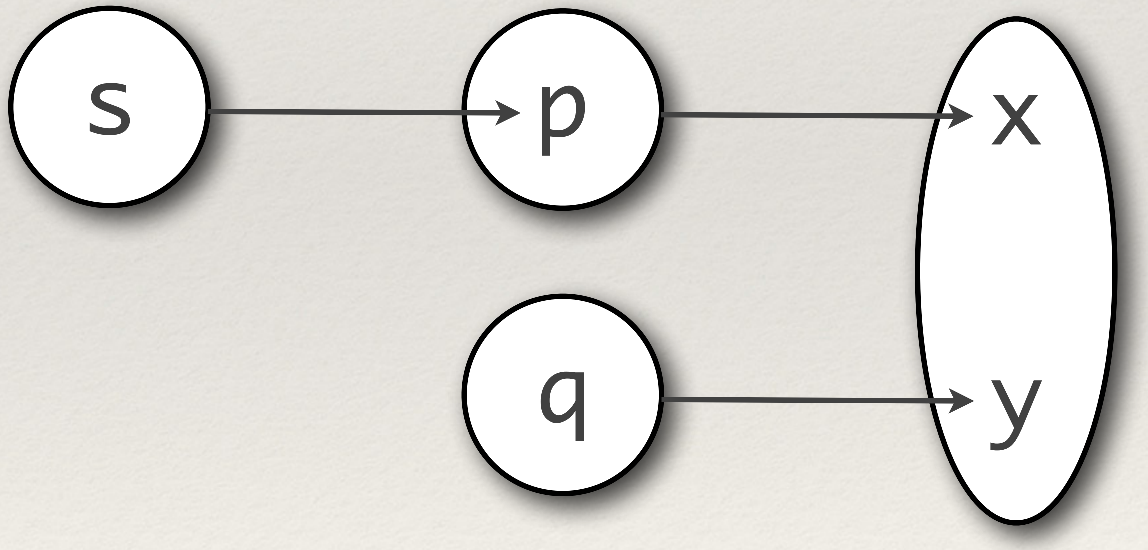
```
p = &x
q = &y
p = q
s = &p
```



Algorithm Overview

- Each nodes is now a collection of objects
- Each node my point-to at most one node

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



Steensgaard's Algorithm

- Flow insensitive, interprocedural, context insensitive, unification based pointer analysis
- Traverse the instructions from top to bottom in a single pass
- Each object in the graph can point-to at most one other node
- Uses type equality (via unification) to get a near linear time performance

Steensgaard’s Language

Steensgaard's Language

- Represent C as a collection of objects and operations on addresses of these objects
- 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;
```

- A variable may reference either a function or a C object
- The return variable is explicit

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

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

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

Provide some primitive object operations

Algorithm Details

Modeling of Value

- A value may be (or include) a function signature or a pointer to a location
  - **Value**: $\boldsymbol{\alpha} = \boldsymbol{\tau} \times \boldsymbol{\lambda}$
  - **Pointer to location**: $\boldsymbol{\tau} = \mathrm{ref}(\boldsymbol{\alpha}) \mid \perp$
  - **Function Signature**: $\boldsymbol{\lambda} = \mathrm{lam}(\alpha_1,\ldots)(\alpha_k,\ldots) \mid \perp$
- Types represent the points-to graph
  - Each node in the graph is a type
  - An edge $t1 \rightarrow t0$ exists if $t1 = \mathbf{ref}(t1 \times \boldsymbol{\lambda})$
- Space usage is $O(n)$ using this representation

Partial Ordering

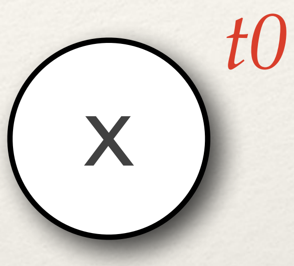
- We define a partial Ordering operator $\trianglelefteq$ such that
  - $(t_1 \trianglelefteq t_2) \Leftrightarrow (t_1 = \perp) \vee (t_1 = t_2)$
  - $(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4)$
- Type of each value is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

Typing Rules

Program

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

Points-to Graph



Type Rules

```
x : t0 = ref(⊥×⊥)
```

- Value types are initialized to ref(⊥×⊥)

Typing Rules

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

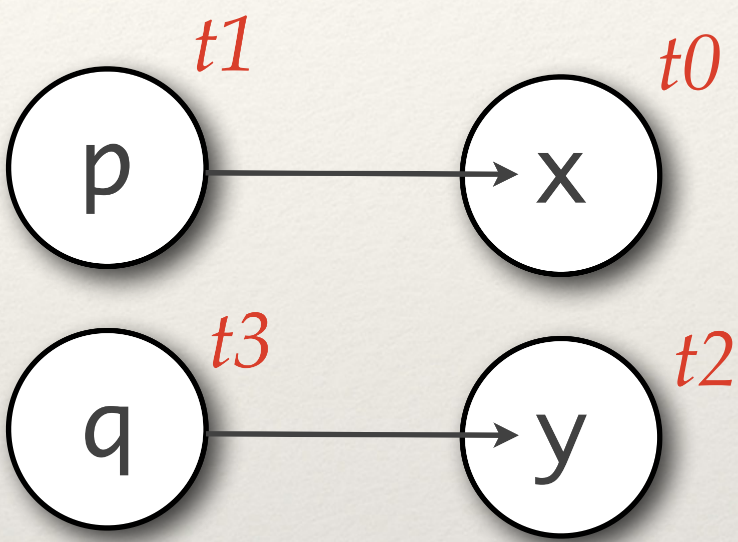
$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \quad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

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

Typing Rules

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

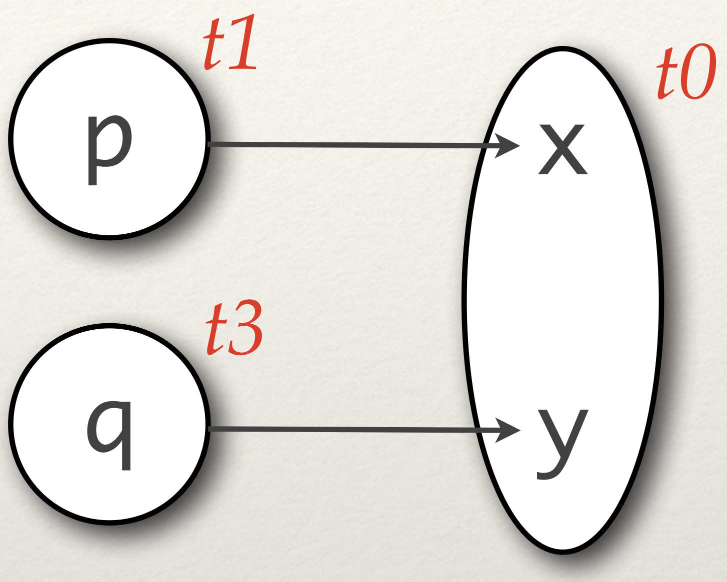
$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \qquad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$




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

Typing Rules

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

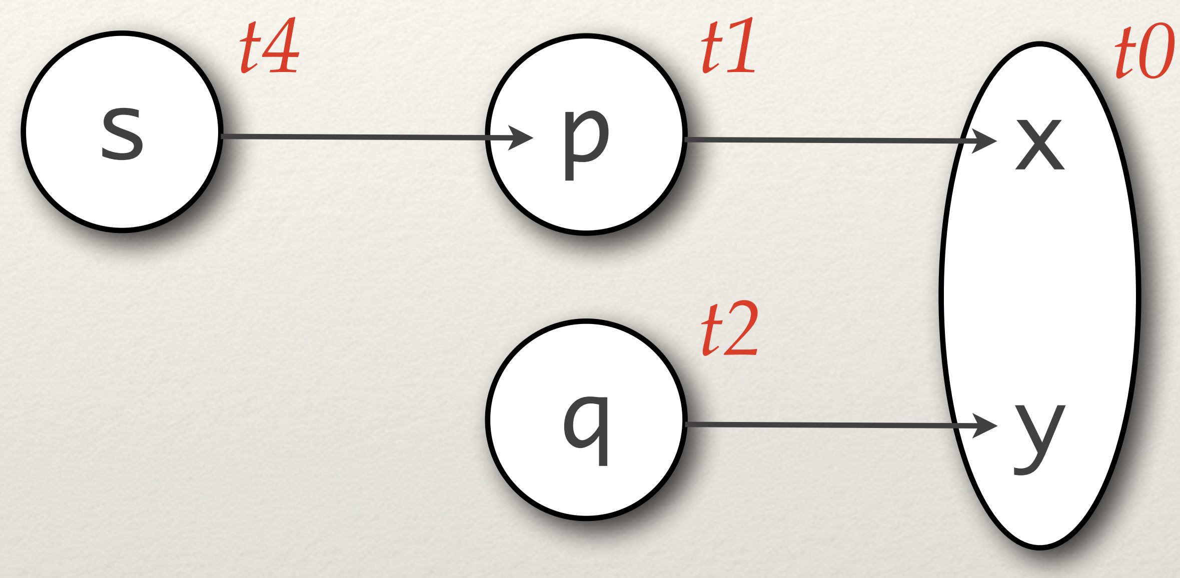


$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

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

Typing Rules

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

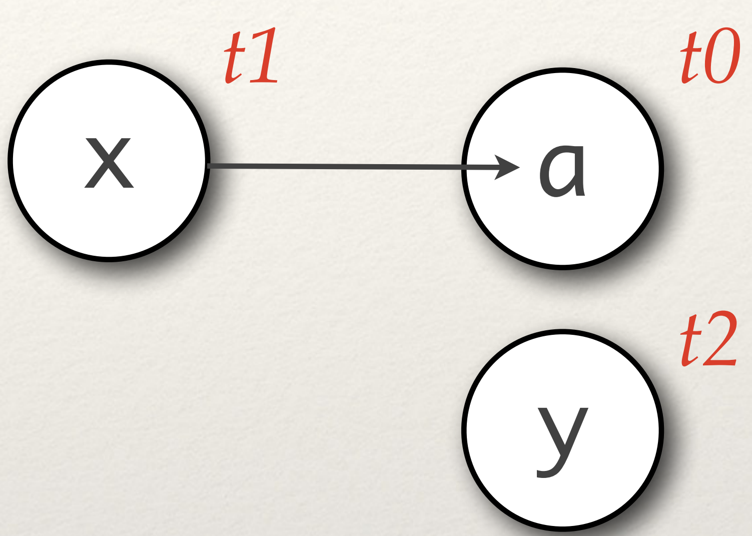


$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \qquad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

```
x : t0 = ref(⊥×⊥)
p : t1 = ref(t0×⊥)
y : t0
q : t3 = ref(t0×⊥)
s : t4 = ref(t1×⊥)
```

Why Partial Ordering Matters

```
x = &a
y = 1
x = y
```




```
a : t0 = ref(⊥×⊥)
x : t1 = ref(t0×⊥)
y : t2 = ref(⊥×⊥)
```

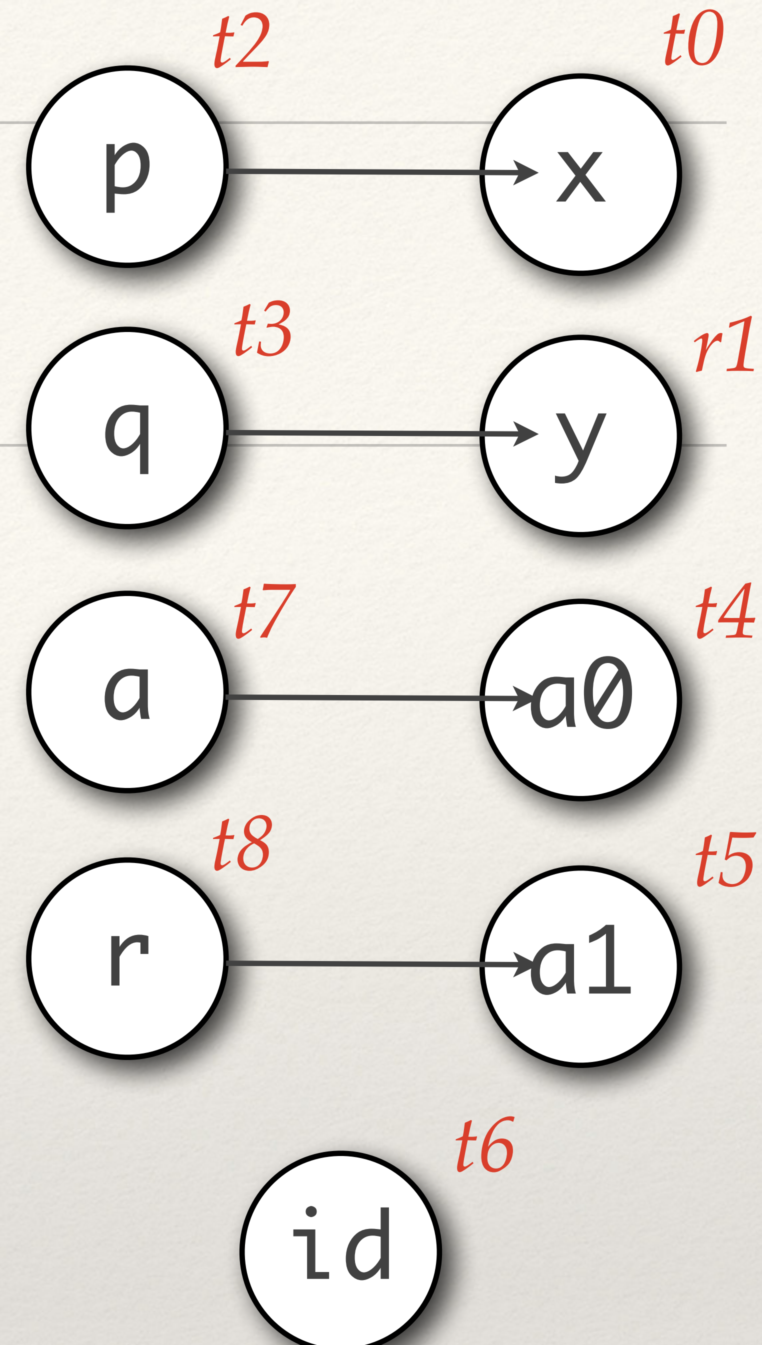
$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \qquad A \vdash \mathsf{y} : \mathbf{ref}(\alpha)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

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

Typing Functions

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

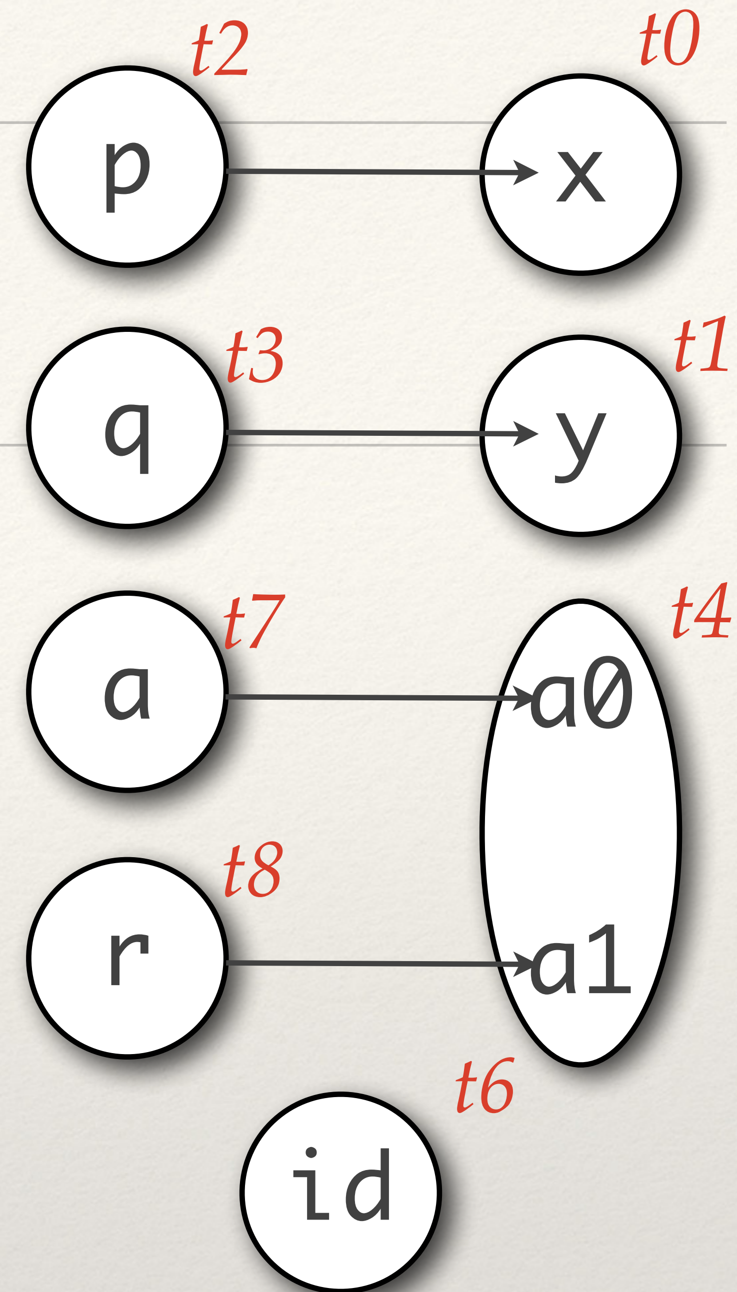


$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{f}_i : \mathbf{ref}(\alpha_i) \\ A \vdash \mathsf{r}_j : \mathbf{ref}(\alpha_{n+j}) \\ \forall s \in S^* : A \vdash \mathit{welltyped}(s)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

Typing Functions

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

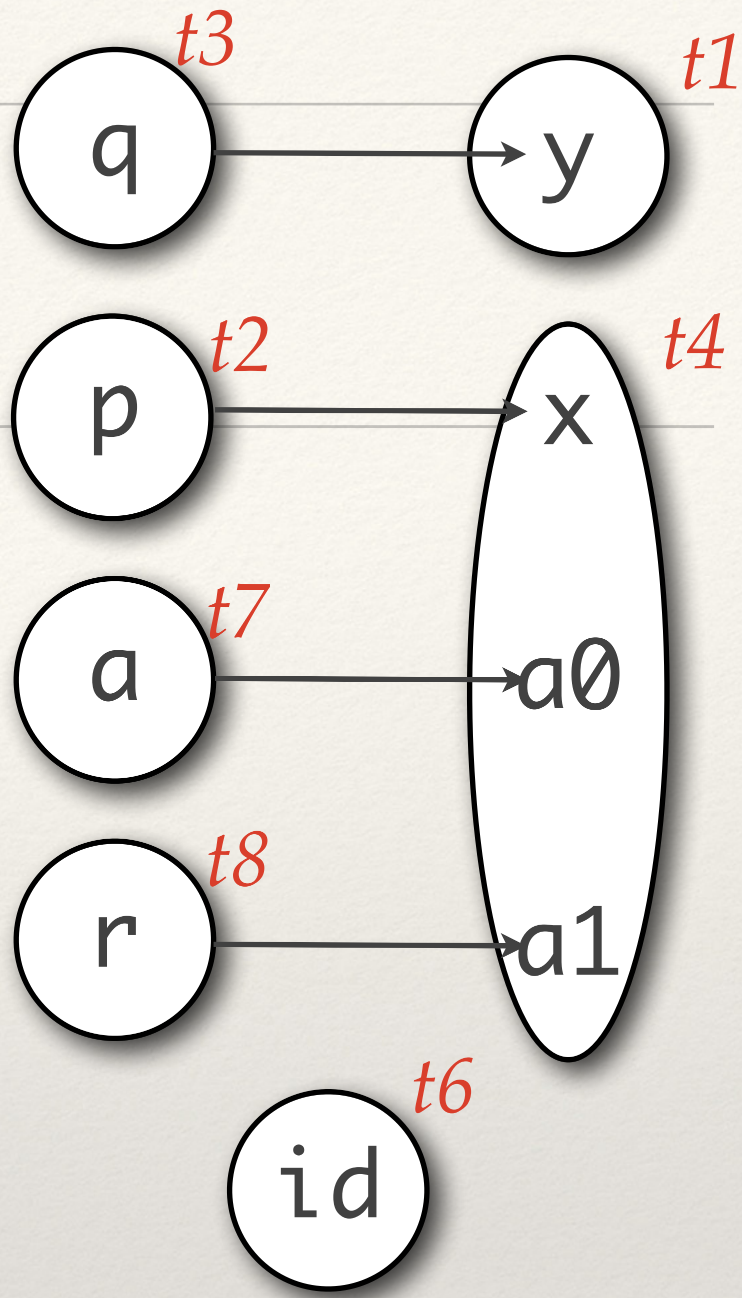
```
x: t0 = ref(⊥×⊥)
y: t1 = ref(⊥×⊥)
p: t2 = ref(t0×⊥)
q: t3 = ref(t1×⊥)
a0:t4 = ref(⊥×⊥)
a1:t4
id:t6 = ref(⊥×lam(t4)(t5))
a :t7 = ref(t4×⊥)
r :t8 = ref(t4×⊥)
```



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

Typing Functions

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



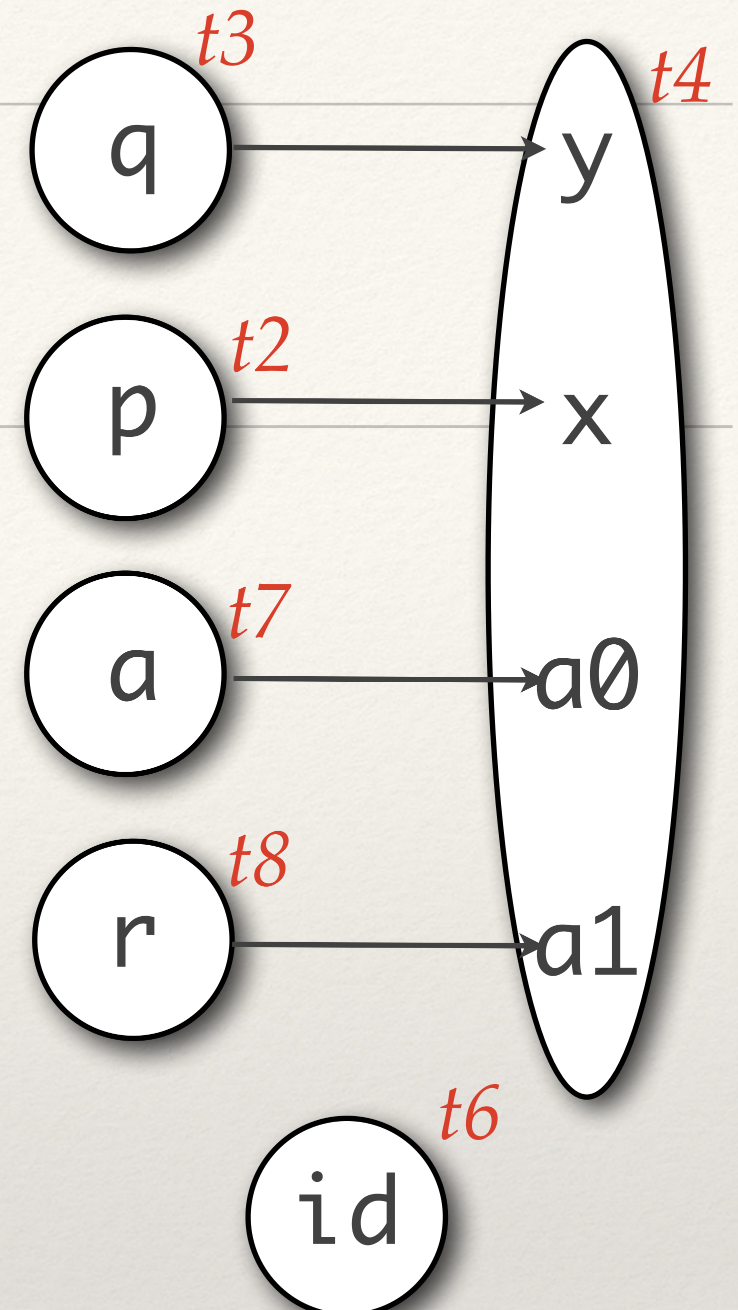

```
x: t4
y: t1 = ref(⊥×⊥)
p: t2 = ref(t4×⊥)
q: t3 = ref(t1×⊥)
a0:t4 = ref(⊥×⊥)
a1:t4
id:t6 = ref(⊥×lam(t4)(t5))
a :t7 = ref(t4×⊥)
r :t8 = ref(t4×⊥)
```

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

Typing Functions

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

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



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

Steensgaard Unifies Function Arguments

- If $t_1 \trianglelefteq C$ and $t_2 \trianglelefteq C$ then $t_1 = t_2$
  - Recall $(t \trianglelefteq C) \Leftrightarrow (t = \bot) \vee (t = C)$
- Function arguments are unified
- Run on real programs, most pointers will alias

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

Results

XXX Results

XXX 157 classes were empty.

446 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	. . .
landi:allroots	18	67																										
landi:assembler	157	446	3		3	1					1																	
landi:loader	90	211	1	1			1	1						1														
landi:compiler	116	166																					1					
landi:simulator	232	464	3	1	2	1			2				2		1								1	2				
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									
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					
spec:li	449	877	1	2	1		2																					
spec:sc	557	1000	26	4	3			1	2						1										1			
spec:alvinn	43	73																										
spec:ear	192	532	3		2			1																			1	
LambdaMOO	1369	2580	9	2	3		1	2																	1			

…

# of vars.	83	. . .	113	. . .	120	. . .	285	. . .	613	. . .	624
landi:allroots											
landi:assembler					1						
landi:loader											
landi:compiler											
landi:simulator	1										
landi:lex315											
landi:football											
austin:anagram											
austin:backprop											
austin:bc											
austin:ft											
austin:ks											
austin:yacr2											
spec:compress											
spec:eqntott											
spec:espresso											
spec:li											1
spec:sc			1				1				
spec:alvinn											
spec:ear											
LambdaMOO									1		

Results

- XXX Todo desctiption of what it means to be in 0 and 1 column
- Why is it good to be in those columns

# of vars.	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
landi:allroots	18	67																			
landi:assembler	157	446	3		3	1					1										
landi:loader	90	211	1	1			1	1						1							
landi:compiler	116	166																			
landi:simulator	232	464	3	1	2	1			2				2		1						
landi:lex315	52	91																			
landi:football	214	570	15	14	1					1											
austin:anagram	54	65					1		1												
austin:backprop	43	69																			
austin:bc	297	551	2													2			2		
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				
spec:li	449	877	1	2	1		2														
spec:sc	557	1000	26	4	3			1	2						1						
spec:alvinn	43	73																			
spec:ear	192	532	3		2			1													
LambdaMOO	1369	2580	9	2	3		1	2													

Unoptimized Pruned Results

# 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
landi:allroots	0	1																																																	
landi:assembler	10	8	3		3	1					1																					1															1				
landi:loader	9	6	1	1			1	1					1																																						
landi:compiler	0	1																					1												1																
landi:simulator	2	4	3	1	2	1			2				2		1								1	2													2							1							
landi:lex315	2																							1																											
landi:football	5		1	1					1																	1																									
austin:anagram	3	3					1		1																																										
austin:backprop	1	9																																																	
austin:bc	5	5	2													2			2																							1									
austin:ft	4	2																																																	
austin:ks	4	1	2			1																																													
austin:yacr2	29	1	3	1																																															
spec:compress	2	4	1																																																
spec:eqntott	5	8	2				1	1	1																																										
spec:espresso	14	19	12	2	2	1	2	1								1	1						1								1			1					1												
spec:li	2	4	1	2	1		2																						1																						
spec:sc	7	10	5	4	2			1	1						1										1																					1				1	
spec:alvinn	1	9																																																	
spec:ear	15	23	3		2			1																			1																								
LambdaMOO	8	15	8	2	3		1	2		1															1										1																1

Optimized Results

# 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	... 9	30	31	32	33	... 44	45	... 52	... 74	... 78	... 83	... 113	... 120	... 285	... 613 ...
landi:allroots																																										
landi:assembler	2																															1								1		
landi:loader	1													1																												
landi:compiler																						1																				
landi:simulator	1	1																				1																1				
landi:lex315	1																						1																			
landi:football	3																																									
austin:anagram	2	1																																								
austin:backprop																																										
austin:bc	2																																				1					
austin:ft	1	1																																								
austin:ks	1					1																																				
austin:yacr2	26																																									
spec:compress			1																																							
spec:eqntott	1	2					1	1																																		
spec:espresso	2	1					1									1	1					1						1								1						
spec:li																																										
spec:sc	1	2																																					1			
spec:alvinn																																										
spec:ear	8	12						1																		1																
LambdaMOO	5	1	1		2																			1																		1

Conclusion

Author's Conclusions

- First algorithm to scale to 100kLOC (previous methods only scaled to 10kLOC)
- Previous scalable methods were not precise
- Previous precise methods were not scalable
- Uses little memory and performs the analysis in a reasonable amount of time on real programs
- Analysis can be used to seed more precise analysis
- Field insensitivity is a major contributor to imprecision

Conclusion

- Uses well studied technique (unification) and applies it to pointer analysis
- There is no distinction between $a = b$ and $b = a$, this makes it less precise than subset based points-to analysis
- Not used in practice in part due to unification of 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)
```

settype(e, t):
  type$(e) \leftarrow t$
  for $x \in$ **pending**(e) do **join**(e, x)

cjoin(e_1, e_2):
  if **type**$(e_2) = \bot$ then
    pending$(e_2) \leftarrow \{e_1\} \cup$ **pending**(e_2)
  else
    join(e_1, e_2)

unify(**ref**$(\tau_1 \times \lambda_1)$, **ref**$(\tau_2 \times \lambda_2)$):
  if $\tau_1 \neq \tau_2$ then **join**(τ_1, τ_2)
  if $\lambda_1 \neq \lambda_2$ then **join**(λ_1, λ_2)

unify(**lam**$(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})$,
    lam$(\alpha'_1 \ldots \alpha'_n)(\alpha'_{n+1} \ldots \alpha'_{n+m})$)
  for $i \in [1 \ldots (n+m)]$ do
  let $\tau_1 \times \lambda_1 = \alpha_i$
    $\tau_2 \times \lambda_2 = \alpha'_i$ in
   if $\tau_1 \neq \tau_2$ then **join**(τ_1, τ_2)
   if $\lambda_1 \neq \lambda_2$ then **join**(λ_1, λ_2)

join(e_1, e_2):
  let $t_1 =$ **type**(e_1)
    $t_2 =$ **type**(e_2)
    $e =$ ecr-union(e_1, e_2) in
   if $t_1 = \bot$ then
    type$(e) \leftarrow t_2$
    if $t_2 = \bot$ then
     pending$(e) \leftarrow$ **pending**$(e_1) \cup$
              pending(e_2)
    else
     for $x \in$ **pending**(e_1) do **join**(e, x)
   else
    type$(e) \leftarrow t_1$
    if $t_2 = \bot$ then
     for $x \in$ **pending**(e_2) do **join**(e, x)
    else
     unify(t_1, t_2)

Unification

Flow Sensitive vs Insensitive Analysis

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

$$p \to y \qquad x$$

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

$$p \to y, \quad p \to x$$

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

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;
```

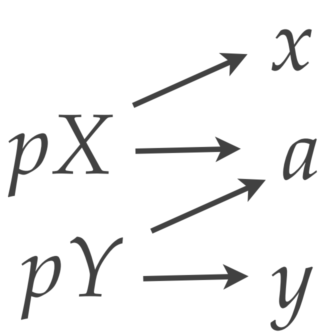
Flow Insensitive: Pointer analysis ignores the order of statements in the program.

$$p1 \longrightarrow y$$

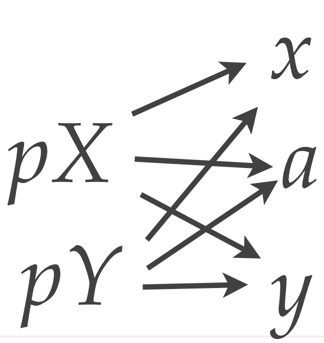
$$p0 \longrightarrow x$$

Context Sensitive vs Insensitive Analysis

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



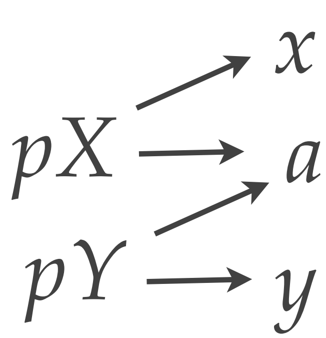
Context Insensitive: Produces spurious aliases.




```
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);
}
```

Union Find

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



Context Insensitive: Produces spurious aliases.