

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
```

.....

Copy 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
```

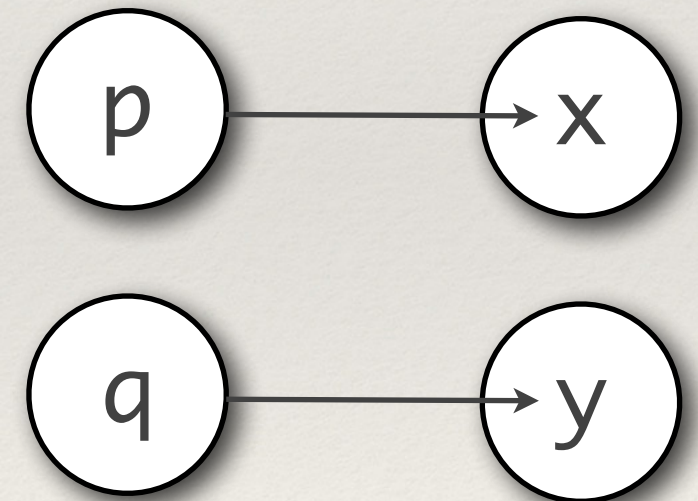
Points-to Graph



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 $q \rightarrow y$

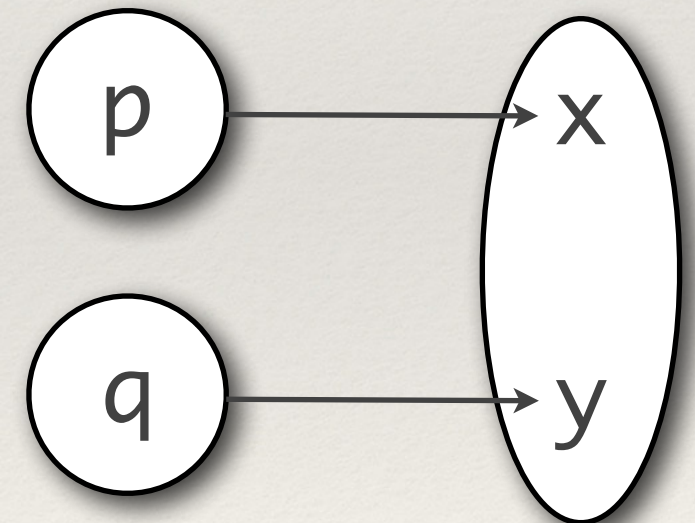
▶
p = &x
q = &y
p = q
s = &p



Algorithm Overview

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

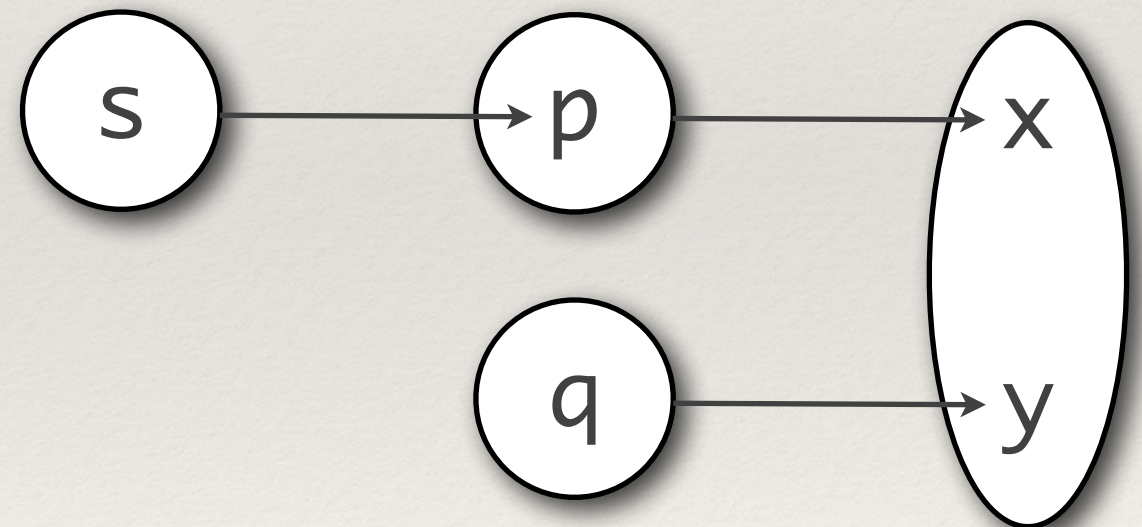
```
p = &x  
q = &y  
p = q  
s = &p
```



Algorithm Overview

- ❖ Each nodes is now a collection of objects
- ❖ Each node may 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

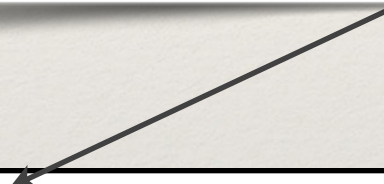
Translating into Steensgaard's Language

```
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:** $\alpha = \tau \times \lambda$
 - ❖ **Pointer to location:** $\tau = \text{ref}(\alpha) \mid \perp$
 - ❖ **Function Signature:** $\lambda = \text{lam}(\alpha_1, \dots)(\alpha_k, \dots) \mid \perp$
- ❖ Types represent the points-to graph
 - ❖ Each node in the graph is a type
 - ❖ An edge $t1 \rightarrow t0$ exists if $t1 = \mathbf{ref}(t1 \times \lambda)$
- ❖ Space usage is $O(n)$ using this representation

Partial Ordering

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

Typing the Program

Program

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

Points-to Graph



Type Rules

```
x : t0 = ref( $\perp \times \perp$ )
```

- ❖ Value types are initialized to $\text{ref}(\perp \times \perp)$

Typing the Program

▶
p = &x
q = &y
p = q
s = &p

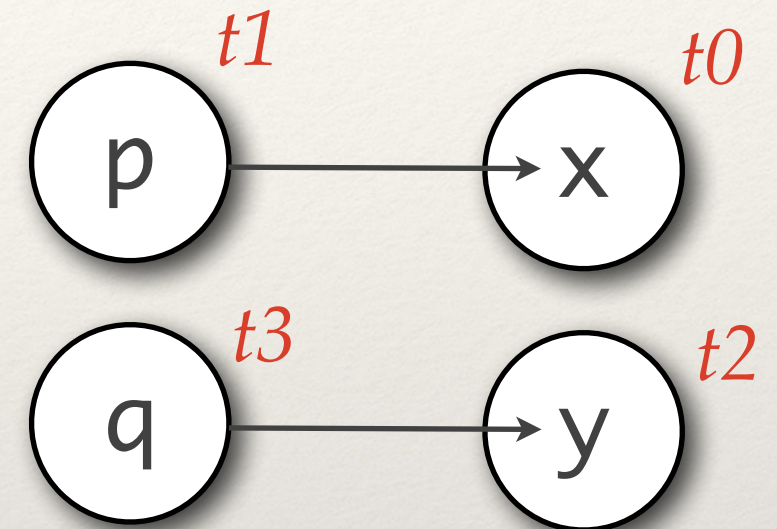


$$\frac{A \vdash x : \mathbf{ref}(\tau \times _) \quad A \vdash y : \tau}{A \vdash \mathit{welltyped}(x = \&y)}$$

x : t0 = ref($\perp \times \perp$)
p : t1 = ref(t0 $\times \perp$)

Typing the Program

▶
p = &x
q = &y
p = q
s = &p

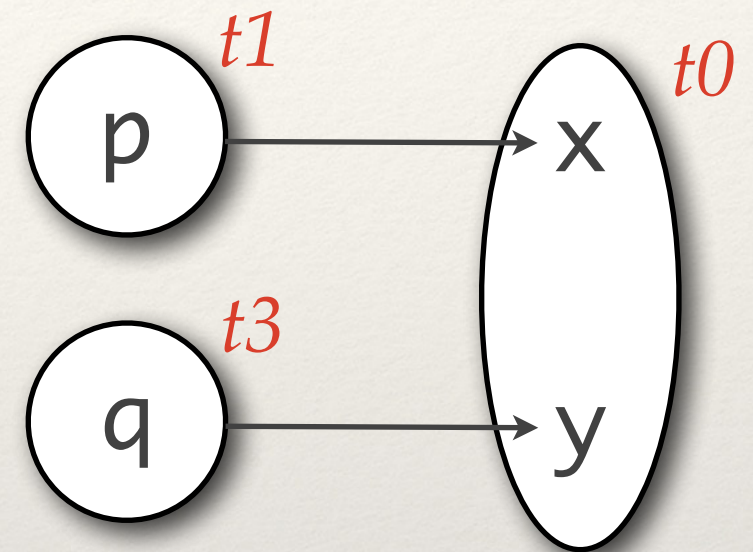


x : t0 = ref(\perp x \perp)
p : t1 = ref(t0x \perp)
y : t2 = ref(\perp x \perp)
q : t3 = ref(t2x \perp)

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

Typing the Program

$p = \&x$
 $q = \&y$
 $p = q$
 $s = \&p$

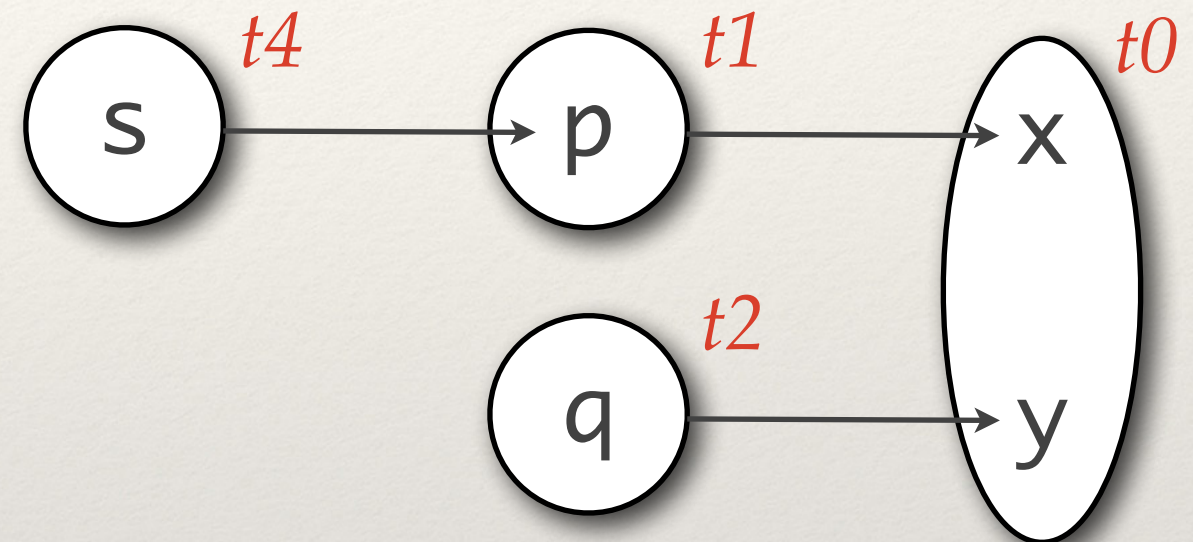


$$\frac{
 \begin{array}{l}
 A \vdash x : \mathbf{ref}(\alpha_1) \\
 A \vdash y : \mathbf{ref}(\alpha_2) \\
 \alpha_2 \trianglelefteq \alpha_1
 \end{array}
 }{
 A \vdash \mathit{welltyped}(x = y)
 }$$

$x : t0 = \mathbf{ref}(\perp x \perp)$
 $p : t1 = \mathbf{ref}(t0x\perp)$
 $y : t0$
 $q : t3 = \mathbf{ref}(t0x\perp)$

Typing the Program

$p = \&x$
 $q = \&y$
 $p = q$
 $s = \&p$



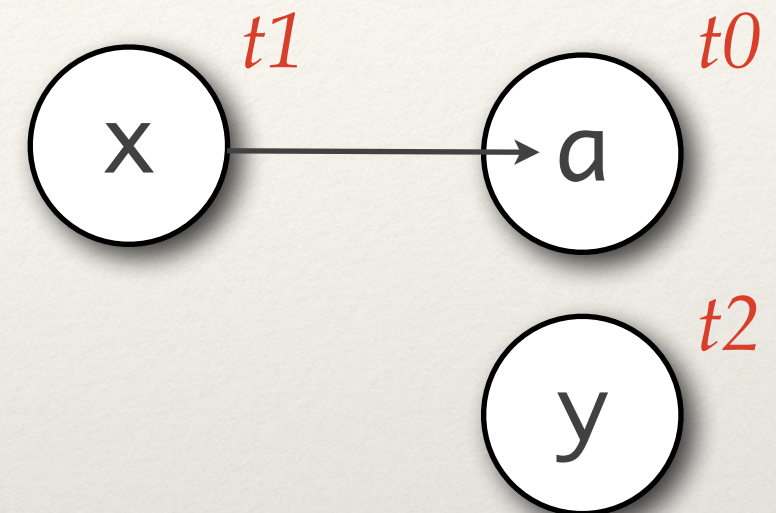
$$\frac{
 \begin{array}{c}
 A \vdash x : \mathbf{ref}(\tau \times _) \\
 A \vdash y : \tau
 \end{array}
 }{
 A \vdash \mathit{welltyped}(x = \&y)
 }$$

$x : t0 = \mathbf{ref}(\perp x \perp)$
 $p : t1 = \mathbf{ref}(t0 x \perp)$
 $y : t0$
 $q : t3 = \mathbf{ref}(t0 x \perp)$
 $s : t4 = \mathbf{ref}(t1 x \perp)$

Why Partial Ordering Matters

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

```
a : t0 = ref( $\perp$ x $\perp$ )  
x : t1 = ref(t0x $\perp$ )  
y : t2 = ref( $\perp$ x $\perp$ )
```


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

Would incorrectly alias x and y

Typing Functions

```

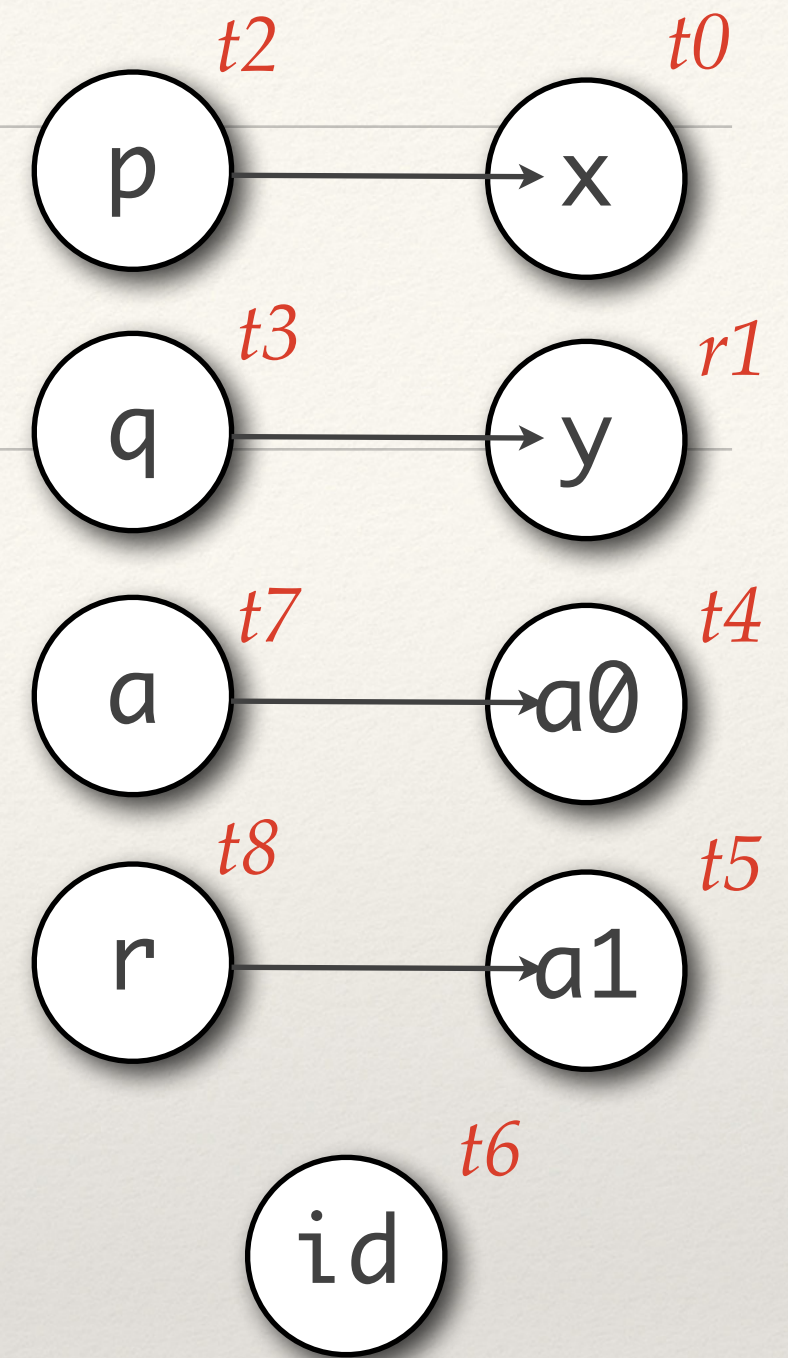
p = &x
q = &y
▶ id = fun(a) -> (r)
    r = a
id(p)
id(q)

```

```

x: t0 = ref( $\perp \times \perp$ )
y: t1 = ref( $\perp \times \perp$ )
p: t2 = ref(t0  $\times \perp$ )
q: t3 = ref(t1  $\times \perp$ )
a0: t4 = ref( $\perp \times \perp$ )
a1: t5 = ref( $\perp \times \perp$ )
id: t6 = ref( $\perp \times \text{lam}(t4)(t5)$ )
a : t7 = ref(t4  $\times \perp$ )
r : t8 = ref(t5  $\times \perp$ )

```



$$\frac{
 \begin{array}{l}
 A \vdash x : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \\
 A \vdash f_i : \mathbf{ref}(\alpha_i) \\
 A \vdash r_j : \mathbf{ref}(\alpha_{n+j}) \\
 \forall s \in S^* : A \vdash \text{welltyped}(s)
 \end{array}
 }{
 A \vdash \text{welltyped}(x = \text{fun}(f_1 \dots f_n) \rightarrow (r_1 \dots r_m) S^*)
 }$$

Typing Functions

```

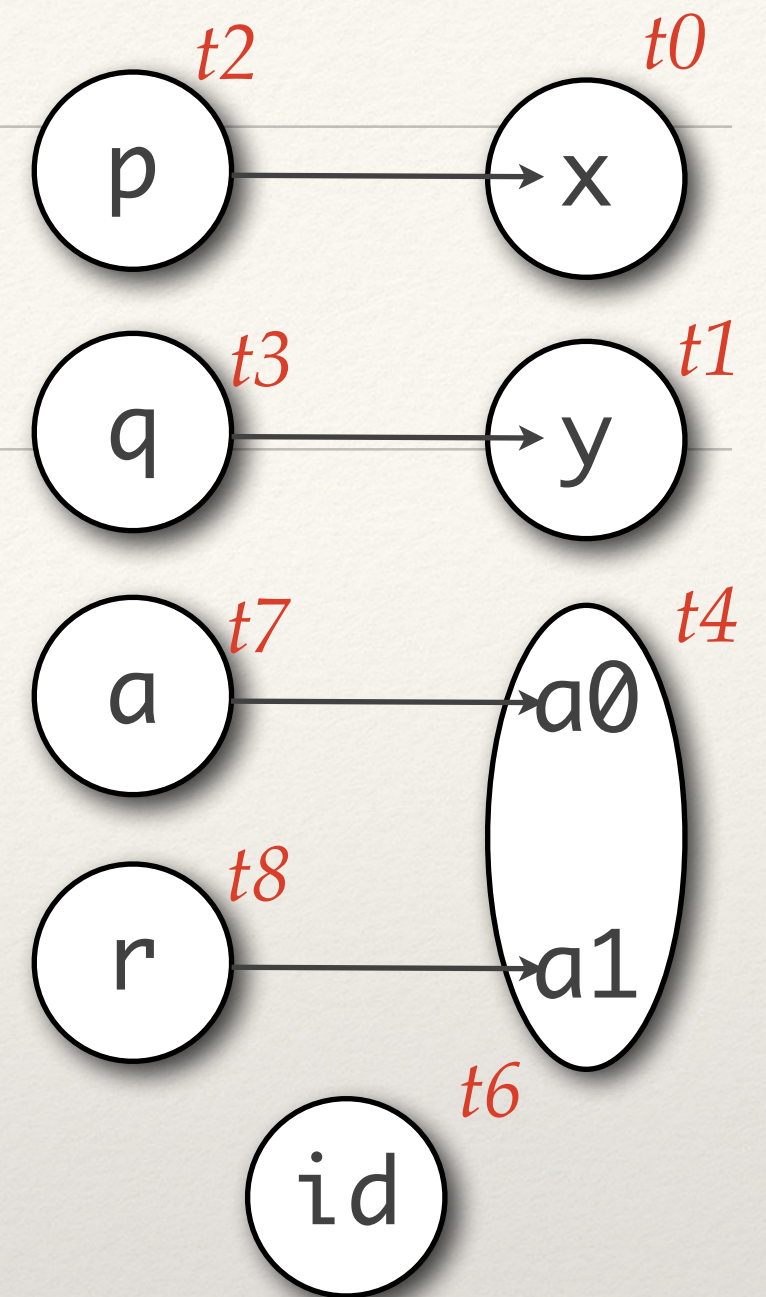
p = &x
q = &y
id = fun(a) -> (r)
    r = a
id(p)
id(q)

```

```

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

```



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

Typing Functions

```

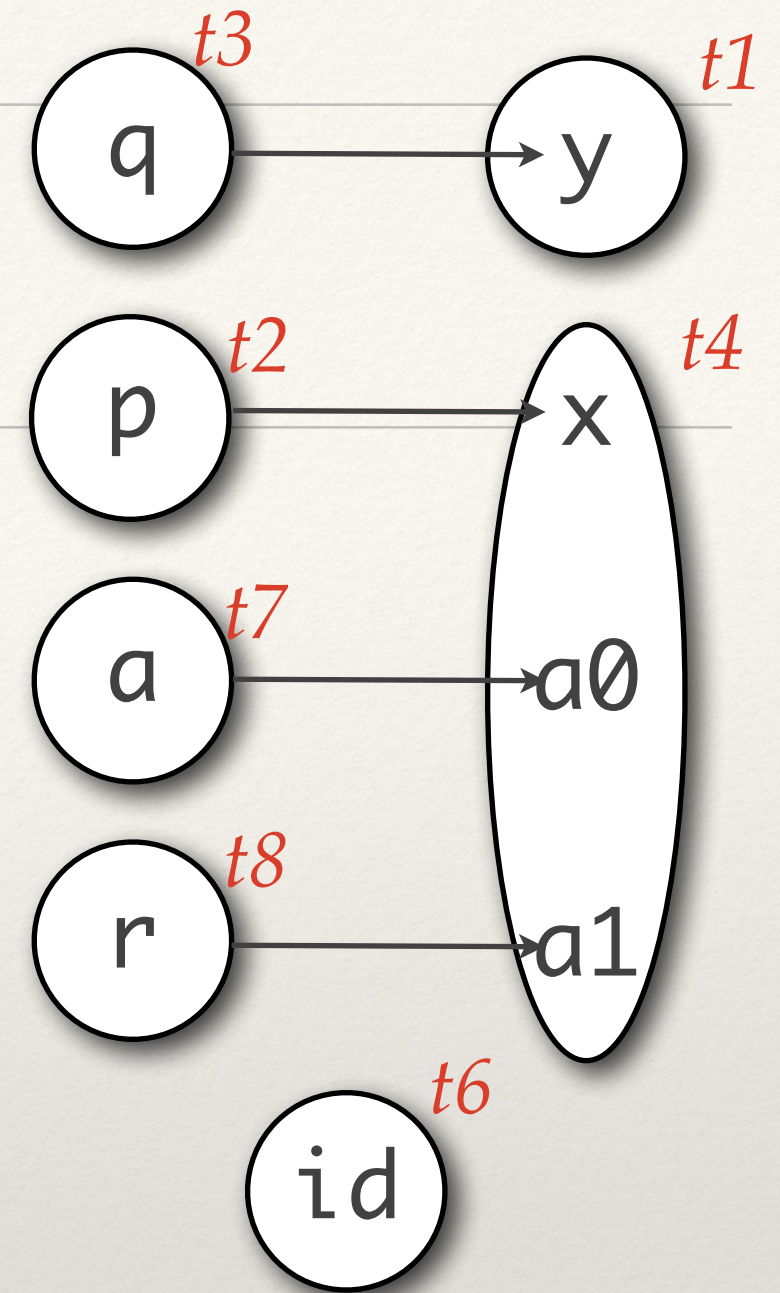
p = &x
q = &y
id = fun(a) -> (r)
    r = a
id(p)
id(q)

```

```

x: t4
y: t1 = ref( $\perp \times \perp$ )
p: t2 = ref(t4  $\times \perp$ )
q: t3 = ref(t1  $\times \perp$ )
a0: t4 = ref( $\perp \times \perp$ )
a1: t4
id: t6 = ref( $\perp \times \text{lam}(t4)(t5)$ )
a : t7 = ref(t4  $\times \perp$ )
r : t8 = ref(t4  $\times \perp$ )

```



$$\begin{array}{c}
 A \vdash x_j : \mathbf{ref}(\alpha'_{n+j}) \\
 A \vdash p : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \\
 A \vdash y_i : \mathbf{ref}(\alpha'_i) \\
 \forall i \in [1 \dots n] : \alpha'_i \sqsubseteq \alpha_i \\
 \forall j \in [1 \dots m] : \alpha_{n+j} \sqsubseteq \alpha'_{n+j} \\
 \hline
 A \vdash \text{welltyped}(x_1 \dots x_m = p(y_1 \dots y_n))
 \end{array}$$

Typing Functions

```

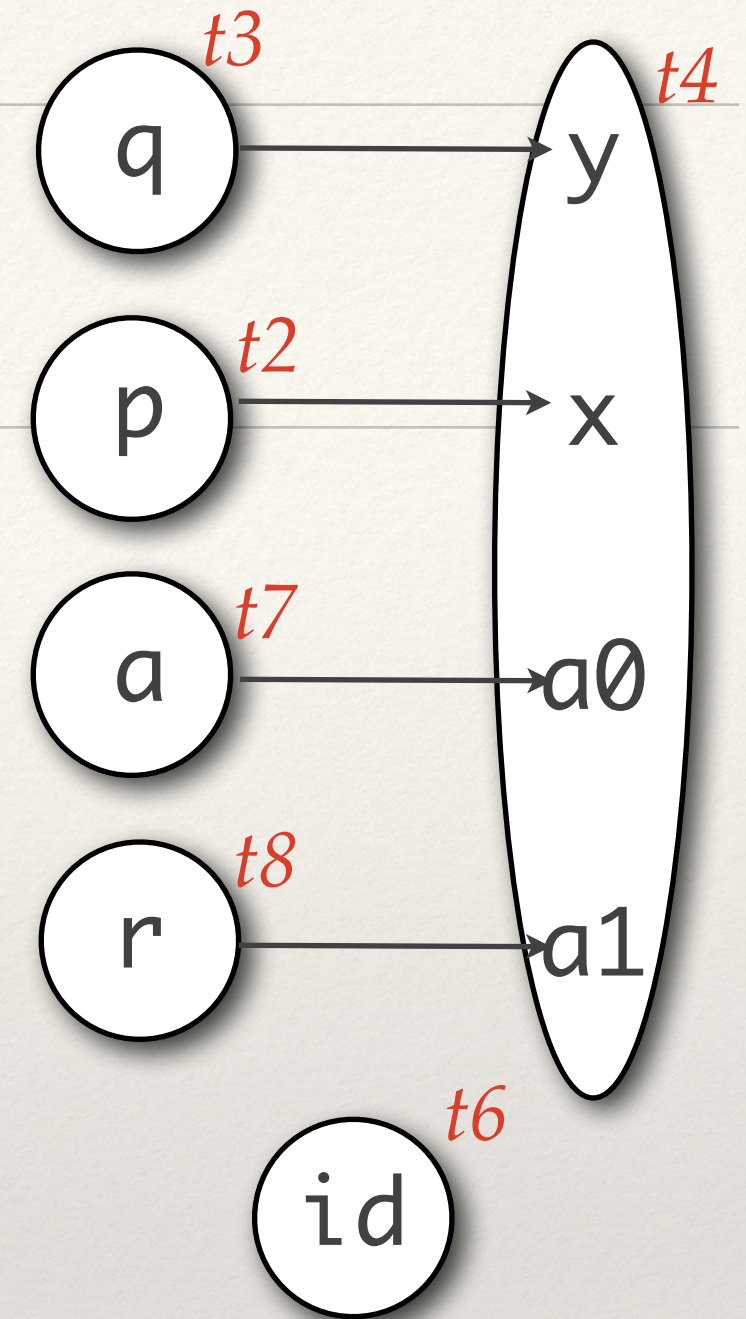
p = &x
q = &y
id = fun(a) -> (r)
    r = a
id(p)
id(q)

```

```

x: t4
y: t4
p: t2 = ref(t4x⊥)
q: t3 = ref(t4x⊥)
a0:t4 = ref(⊥x⊥)
a1:t4
id:t6 = ref(⊥xlam(t4)(t5))
a :t7 = ref(t4x⊥)
r :t8 = ref(t4x⊥)

```



$$\frac{
 \begin{array}{c}
 A \vdash x_j : \mathbf{ref}(\alpha'_{n+j}) \\
 A \vdash p : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \\
 A \vdash y_i : \mathbf{ref}(\alpha'_i) \\
 \forall i \in [1 \dots n] : \alpha'_i \sqsubseteq \alpha_i \\
 \forall j \in [1 \dots m] : \alpha_{n+j} \sqsubseteq \alpha'_{n+j}
 \end{array}
 }{
 A \vdash \mathbf{welltyped}(x_1 \dots x_m = p(y_1 \dots y_n))
 }$$

Steensgaard Unifies Function Arguments

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

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


Results

Results

- ❖ Most type variables describe only a single program variable
- ❖ Type variables describing hundreds of program variables are due to global values being unified.

# 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													

Pruned Results

Prune variables which are never pointed to in the program

# 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
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

Eliminate all variables whose address is never taken

# 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

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

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

$$\frac{A \vdash x : \mathbf{ref}(\alpha_1) \quad A \vdash y : \mathbf{ref}(\mathbf{ref}(\alpha_2) \times _) \quad \alpha_2 \sqsubseteq \alpha_1}{A \vdash \mathbf{welltyped}(x = *y)}$$

$$\frac{A \vdash x : \mathbf{ref}(\alpha) \quad A \vdash y_i : \mathbf{ref}(\alpha_i) \quad \forall i \in [1 \dots n] : \alpha_i \sqsubseteq \alpha}{A \vdash \mathbf{welltyped}(x = \mathbf{op}(y_1 \dots y_n))}$$

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

$$\frac{A \vdash x : \mathbf{ref}(\mathbf{ref}(\alpha_1) \times _) \quad A \vdash y : \mathbf{ref}(\alpha_2) \quad \alpha_2 \sqsubseteq \alpha_1}{A \vdash \mathbf{welltyped}(*x = y)}$$

$$\frac{A \vdash x : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \quad A \vdash f_i : \mathbf{ref}(\alpha_i) \quad A \vdash r_j : \mathbf{ref}(\alpha_{n+j}) \quad \forall s \in S^* : A \vdash \mathbf{welltyped}(s)}{A \vdash \mathbf{welltyped}(x = \mathbf{fun}(f_1 \dots f_n) \rightarrow (r_1 \dots r_m) S^*)}$$

$$\frac{A \vdash x_j : \mathbf{ref}(\alpha'_{n+j}) \quad A \vdash p : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \quad A \vdash y_i : \mathbf{ref}(\alpha'_i) \quad \forall i \in [1 \dots n] : \alpha'_i \sqsubseteq \alpha_i \quad \forall j \in [1 \dots m] : \alpha_{n+j} \sqsubseteq \alpha'_{n+j}}{A \vdash \mathbf{welltyped}(x_1 \dots x_m = p(y_1 \dots y_n))}$$

Type Rules

Implementation

```

x = y:
  let ref( $\tau_1 \times \lambda_1$ ) = type(ecr(x))
  ref( $\tau_2 \times \lambda_2$ ) = type(ecr(y)) in
  if  $\tau_1 \neq \tau_2$  then cjoin( $\tau_1, \tau_2$ )
  if  $\lambda_1 \neq \lambda_2$  then cjoin( $\lambda_1, \lambda_2$ )

x = &y:
  let ref( $\tau_1 \times \_$ ) = type(ecr(x))
   $\tau_2 = \text{ecr}(y)$  in
  if  $\tau_1 \neq \tau_2$  then join( $\tau_1, \tau_2$ )

x = *y:
  let ref( $\tau_1 \times \lambda_1$ ) = type(ecr(x))
  ref( $\tau_2 \times \_$ ) = type(ecr(y)) in
  if type( $\tau_2$ ) =  $\perp$  then
    settype( $\tau_2$ , ref( $\tau_1 \times \lambda_1$ ))
  else
    let ref( $\tau_3 \times \lambda_3$ ) = type( $\tau_2$ ) in
    if  $\tau_1 \neq \tau_3$  then cjoin( $\tau_1, \tau_3$ )
    if  $\lambda_1 \neq \lambda_3$  then cjoin( $\lambda_1, \lambda_3$ )

x = op( $y_1 \dots y_n$ ):
  for  $i \in [1 \dots n]$  do
    let ref( $\tau_1 \times \lambda_1$ ) = type(ecr(x))
    ref( $\tau_2 \times \lambda_2$ ) = type(ecr( $y_i$ )) in
    if  $\tau_1 \neq \tau_2$  then cjoin( $\tau_1, \tau_2$ )
    if  $\lambda_1 \neq \lambda_2$  then cjoin( $\lambda_1, \lambda_2$ )

x = allocate(y):
  let ref( $\tau \times \_$ ) = type(ecr(x)) in
  if type( $\tau$ ) =  $\perp$  then
    let [ $e_1, e_2$ ] = MakeECR(2) in
    settype( $\tau$ , ref( $e_1 \times e_2$ )))

*x = y:
  let ref( $\tau_1 \times \_$ ) = type(ecr(x))
  ref( $\tau_2 \times \lambda_2$ ) = type(ecr(y))
  if type( $\tau_1$ ) =  $\perp$  then
    settype( $\tau_1$ , ref( $\tau_2 \times \lambda_2$ ))
  else
    let ref( $\tau_3 \times \lambda_3$ ) = type( $\tau_1$ ) in
    if  $\tau_2 \neq \tau_3$  then cjoin( $\tau_3, \tau_2$ )
    if  $\lambda_2 \neq \lambda_3$  then cjoin( $\lambda_3, \lambda_2$ )
  
```

```

x = fun( $f_1 \dots f_n$ )  $\rightarrow$  ( $r_1 \dots r_m$ )  $S^{\infty}$ :
  let ref( $\_ \times \lambda$ ) = type(ecr(x))
  if type( $\lambda$ ) =  $\perp$  then
    settype( $\lambda$ , lam( $\alpha_1 \dots \alpha_n$ )( $\alpha_{n+1} \dots \alpha_{n+m}$ ))
  where
    ref( $\alpha_i$ ) = type(ecr( $f_i$ )), for  $i \leq n$ 
    ref( $\alpha_i$ ) = type(ecr( $r_{i-n}$ )), for  $i > n$ 
  else
    let lam( $\alpha_1 \dots \alpha_n$ )( $\alpha_{n+1} \dots \alpha_{n+m}$ ) = type( $\lambda$ ) in
    for  $i \in [1 \dots n]$  do
      let  $\tau_1 \times \lambda_1 = \alpha_i$ 
      ref( $\tau_2 \times \lambda_2$ ) = type(ecr( $f_i$ )) in
      if  $\tau_1 \neq \tau_2$  then join( $\tau_2, \tau_1$ )
      if  $\lambda_1 \neq \lambda_2$  then join( $\lambda_2, \lambda_1$ )
    for  $i \in [1 \dots m]$  do
      let  $\tau_1 \times \lambda_1 = \alpha_{n+i}$ 
      ref( $\tau_2 \times \lambda_2$ ) = type(ecr( $r_i$ )) in
      if  $\tau_1 \neq \tau_2$  then join( $\tau_1, \tau_2$ )
      if  $\lambda_1 \neq \lambda_2$  then join( $\lambda_1, \lambda_2$ )

 $x_1 \dots x_m = p(y_1 \dots y_n)$ :
  let ref( $\_ \times \lambda$ ) = type(ecr(p)) in
  if type( $\lambda$ ) =  $\perp$  then
    settype( $\lambda$ , lam( $\alpha_1 \dots \alpha_n$ )( $\alpha_{n+1} \dots \alpha_{n+m}$ ))
  where
     $\alpha_i = \tau_i \times \lambda_i$ 
    [ $\tau_i, \lambda_i$ ] = MakeECR(2)
  let lam( $\alpha_1 \dots \alpha_n$ )( $\alpha_{n+1} \dots \alpha_{n+m}$ ) = type( $\lambda$ ) in
  for  $i \in [1 \dots n]$  do
    let  $\tau_1 \times \lambda_1 = \alpha_i$ 
    ref( $\tau_2 \times \lambda_2$ ) = type(ecr( $y_i$ )) in
    if  $\tau_1 \neq \tau_2$  then cjoin( $\tau_1, \tau_2$ )
    if  $\lambda_1 \neq \lambda_2$  then cjoin( $\lambda_1, \lambda_2$ )
  for  $i \in [1 \dots m]$  do
    let  $\tau_1 \times \lambda_1 = \alpha_{n+i}$ 
    ref( $\tau_2 \times \lambda_2$ ) = type(ecr( $x_i$ )) in
    if  $\tau_1 \neq \tau_2$  then cjoin( $\tau_2, \tau_1$ )
    if  $\lambda_1 \neq \lambda_2$  then cjoin( $\lambda_2, \lambda_1$ )
  
```



```

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

```

```

cjoin( $e_1, e_2$ ):
  if type( $e_2$ ) =  $\perp$  then
    pending( $e_2$ )  $\leftarrow \{e_1\} \cup \text{pending}(e_2)$ 
  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( $\tau_1, \tau_2$ )
  if  $\lambda_1 \neq \lambda_2$  then join( $\lambda_1, \lambda_2$ )

```

```

unify(lam( $\alpha_1 \dots \alpha_n$ )( $\alpha_{n+1} \dots \alpha_{n+m}$ ),
      lam( $\alpha'_1 \dots \alpha'_n$ )( $\alpha'_{n+1} \dots \alpha'_{n+m}$ )):
  for  $i \in [1 \dots (n + m)]$  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( $\tau_1, \tau_2$ )
      if  $\lambda_1 \neq \lambda_2$  then join( $\lambda_1, \lambda_2$ )

```

```

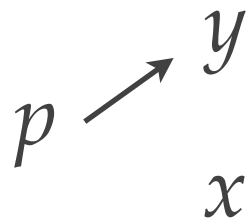
join( $e_1, e_2$ ):
  let  $t_1 = \text{type}(e_1)$ 
   $t_2 = \text{type}(e_2)$ 
   $e = \text{ecr-union}(e_1, e_2)$  in
  if  $t_1 = \perp$  then
    type( $e$ )  $\leftarrow t_2$ 
    if  $t_2 = \perp$  then
      pending( $e$ )  $\leftarrow \text{pending}(e_1) \cup \text{pending}(e_2)$ 
    else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

```

Unification

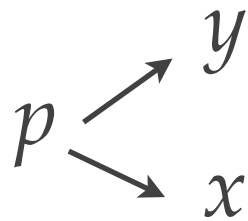
Flow Sensitive vs Insensitive Analysis

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



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

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



SSA Remedies Some Flow Insensitivity

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

$$p1 \longrightarrow y$$
$$p0 \longrightarrow x$$

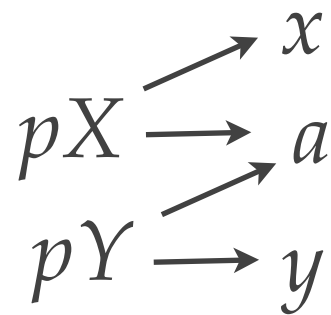
```
x = 1;  
y = 2;  
p0 = &x;  
p1 = &y;
```

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

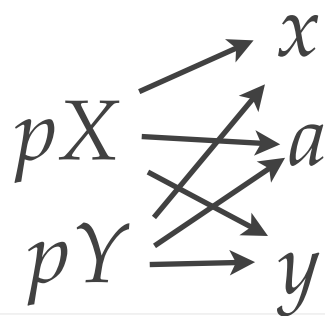
$$p1 \longrightarrow y$$
$$p0 \longrightarrow x$$

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);  
}
```