Bjarne Steensgaard

Points-to Analysis in Almost Linear Time

Presented by **Abdul Dakkak**

Overview

- Algorithm Overview
- Language
- 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 $g \rightarrow a$ is a substitution that makes the two terms equal
- For Steensgard, unification means union

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 $p \to x$

Program

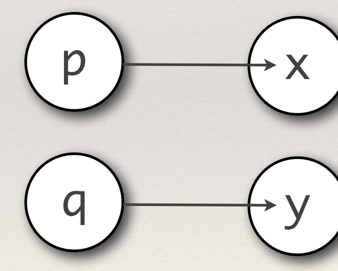
```
p = &x
q = &y
p = q
s = &p
```

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 $q \rightarrow y$

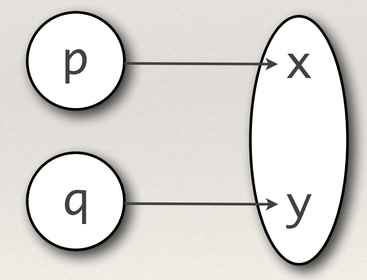
```
p = &x
q = &y
p = q
s = &p
```



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

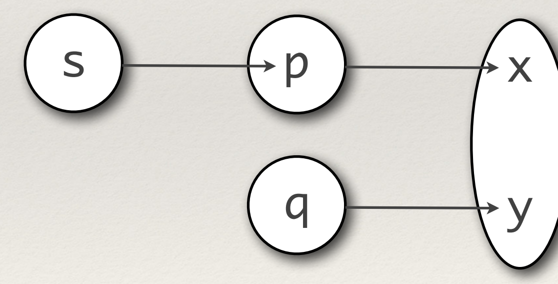
```
p = &x
q = &y
p = q
s = &p
```



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**: $\alpha = \tau \times \lambda$
  - **Pointer to location**: $\tau = \mathrm{ref}(\alpha) \mid \bot$
  - **Function Signature**: $\lambda = \mathrm{lam}(\alpha_1,\ldots)(\alpha_k,\ldots) \mid \bot$
- 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

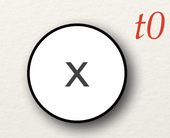
- We define a partial Ordering operator $\trianglelefteq$ such that
  - $(t_1 \trianglelefteq t_2) \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$
  - $(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4)$
- Type of each value is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

Typing the Program

Program

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

Points-to Graph



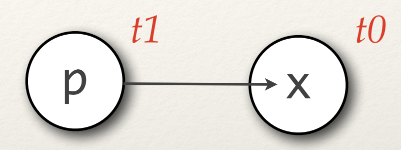
Type Rules

```
x : t0 = ref(⊥×⊥)
```

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

Typing the Program

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



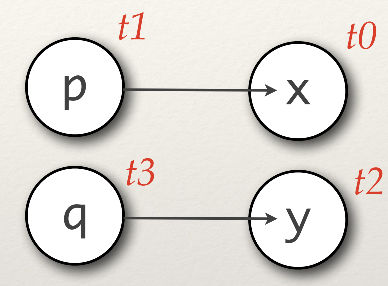
$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \qquad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

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

Typing the Program

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

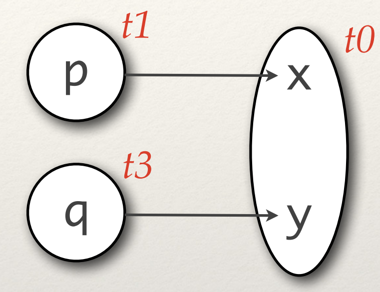
$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \\ A \vdash \mathsf{y} : \tau \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$




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

Typing the Program

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

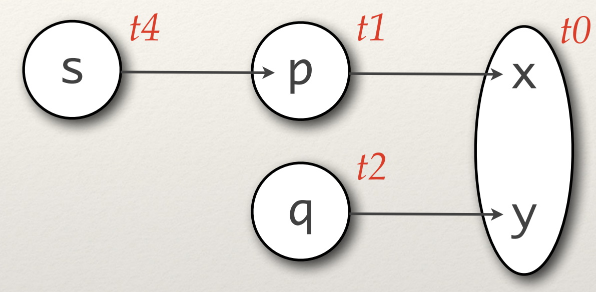


$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

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

Typing the Program

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



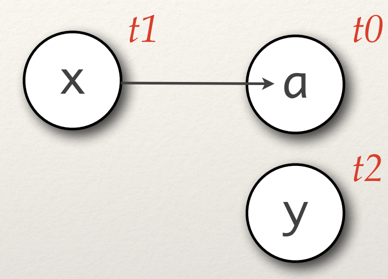
$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \quad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

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

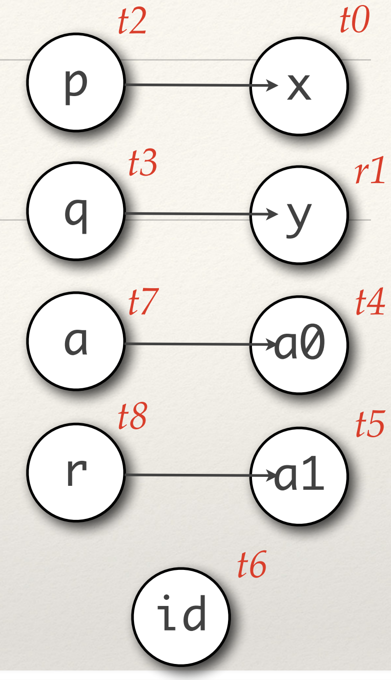


$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

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

Typing Functions

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



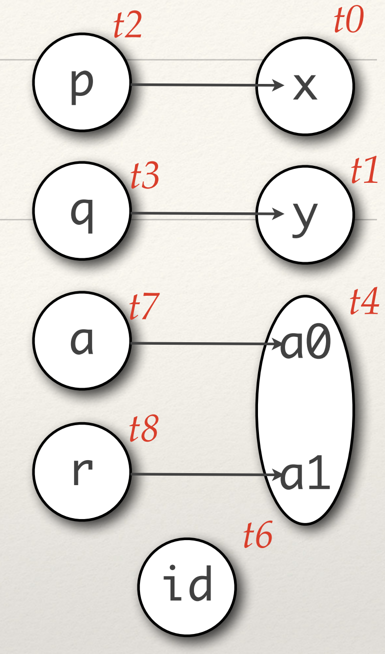

```
x: t0 = ref(⊥×⊥)
y: t1 = ref(⊥×⊥)
p: t2 = ref(t0×⊥)
q: t3 = ref(t1×⊥)
a0:t4 = ref(⊥×⊥)
a1:t5 = ref(⊥×⊥)
id:t6 = ref(⊥×lam(t4)(t5))
a :t7 = ref(t4×⊥)
r :t8 = ref(t5×⊥)
```

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

Typing Functions

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

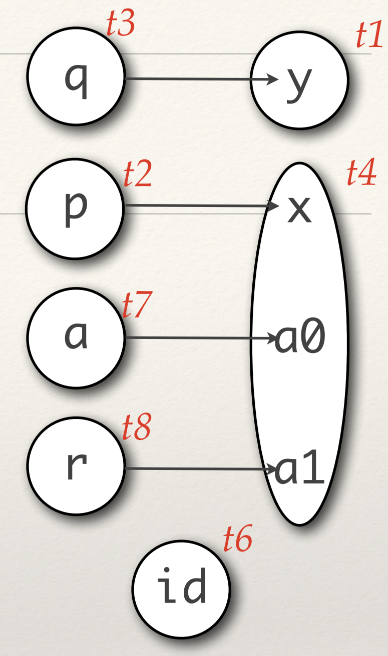
```
x: t0 = ref(⊥×⊥)
y: t1 = ref(⊥×⊥)
p: t2 = ref(t0×⊥)
q: t3 = ref(t1×⊥)
a0:t4 = ref(⊥×⊥)
a1:t4
id:t6 = ref(⊥×lam(t4)(t5))
a :t7 = ref(t4×⊥)
r :t8 = ref(t4×⊥)
```



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

Typing Functions

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

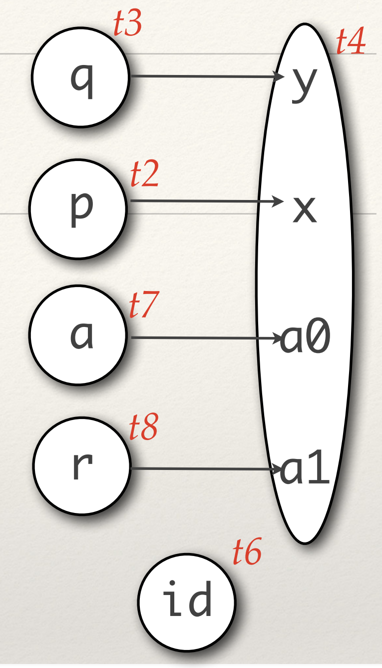



```
x: t4
y: t1 = ref(⊥×⊥)
p: t2 = ref(t4×⊥)
q: t3 = ref(t1×⊥)
a0:t4 = ref(⊥×⊥)
a1:t4
id:t6 = ref(⊥×lam(t4)(t5))
a :t7 = ref(t4×⊥)
r :t8 = ref(t4×⊥)
```

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

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 $(t \trianglelefteq C) \Leftrightarrow (t = \perp) \vee (t = C)$
- Function arguments are unified
- Run on real programs, most pointers will alias

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

Results

Results

157 classes were empty : these correspond to user functions and variables allocated by the runtime.

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 . . .	83 . . .	113 . . .	120 . . .	285 . . .	613 . . .	624
landi:allroots	18	67																															
landi:assembler	157	446	3		3	1					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				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														
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																										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	

...

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								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
landi:allroots																													
landi:assembler	2																												
landi:loader	1													1															
landi:compiler																							1						
landi:simulator	1	1																					1						
landi:lex315	1																							1					
landi:football	3																												
austin:anagram	2	1																											
austin:backprop																													
austin:bc	2																												
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
spec:li																													
spec:sc	1	2																											
spec:alvinn																													
spec:ear	8	12						1																		1			
LambdaMOO	5	1	1		2																				1				

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

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

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

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

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

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

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

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

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

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) ← t
  for x ∈ pending(e) do join(e, x)

cjoin(e1, e2):
  if type(e2) = ⊥ then
    pending(e2) ← {e1} ∪ pending(e2)
  else
    join(e1, e2)

unify(ref(τ1 × λ1), ref(τ2 × λ2)):
  if τ1 ≠ τ2 then join(τ1, τ2)
  if λ1 ≠ λ2 then join(λ1, λ2)

unify(lam(α1 ... αn)(αn+1 ... αn+m),
      lam(α'1 ... α'n)(α'n+1 ... α'n+m))
  for i ∈ [1 ... (n + m)] do
  let τ1 × λ1 = αi
      τ2 × λ2 = α'i in
    if τ1 ≠ τ2 then join(τ1, τ2)
    if λ1 ≠ λ2 then join(λ1, λ2)

join(e1, e2):
  let t1 = type(e1)
      t2 = type(e2)
      e = ecr-union(e1, e2) in
    if t1 = ⊥ then
      type(e) ← t2
      if t2 = ⊥ then
        pending(e) ← pending(e1) ∪
                     pending(e2)
      else
        for x ∈ pending(e1) do join(e, x)
    else
      type(e) ← t1
      if t2 = ⊥ then
        for x ∈ pending(e2) do join(e, x)
      else
        unify(t1, t2)
```

Unification

Union Find Data Structure

Flow Sensitive vs Insensitive Analysis

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

$$p \rightarrow y \qquad x$$

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

$$p \rightarrow y, \quad p \rightarrow x$$

```
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.

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

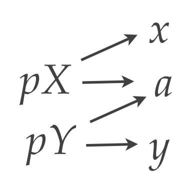
Flow Insensitive: Pointer analysis ignores the order of statements in the program.

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

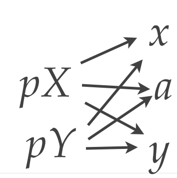
```
x = 1;
y = 2;
p0 = &x;
p1 = &y;
```

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);
}
```

Object Sensitive Analysis

Heap Modeling

Point-to Analysis using Andersen

Point-to Analysis using BDD

Point-to Analysis using CFL Reachability