Bjarne Steensgaard

Points-to Analysis in Almost Linear Time

Presented by **Abdul Dakkak**

Overview

- Definitions
- Steensgaard's Language
- Steensgaard's Algorithm
- 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

Available Expressions: If **p* aliases *a* or *b* then the second computation of *a+b* is not redundant.

```
*p = a + b;
 y = a + b;
```

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

```
 x = 3;
*p = 4;
 g = x;
```

Steensgaard’s Language

Bubble Sort

- Declare a Point structure
- Swap elements if two points are lexicographically less than each other
- In **main**: allocate two lists and perform bubble sort on them

```
typedef struct {
  double x, y;
} Point_t;
bool less(Point_t a, Point_t b) {
  return (a.x < b.x) || (a.x == b.x && a.y < b.y);
}
void swap(Point_t * a, Point_t * b) {
  Point_t tmp = *a;
  *a = *b;
  *b = tmp;
}
void bubbleSort(Point_t *pts, int len) {
  bool swapped = true;
  while (swapped) {
    int ii = 1;
    swapped = false;
    while (ii < len) {
      Point_t * prev = &pts[ii - 1];
      Point_t * curr = &pts[ii];
      if (less(*prev, *curr)) {
        swap(prev, curr);
        swapped = true;
      }
      ii++;
    }
  }
}
int main(void) {
  lenA = 4;
  lenB = 4;
  A = malloc(lenA);
  A = malloc(lenA);
  A = {{0,0}, {0,1}, {1,1}, {1,0}};
  B = {{1,0}, {1,1}, {0,1}, {0,0}};
  bubbleSort(A, lenA);
  bubbleSort(B, lenB);
  return 0;
}
```

Translating into Steensgaard's Language

- Models C language
- C pointer operations can be desugared

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

```
typedef struct {
  double x, y;
} Point_t;
bool less(Point_t a, Point_t b) {
  return (a.x < b.x) || (a.x == b.x && a.y < b.y);
}
void swap(Point_t * a, Point_t * b) {
  Point_t tmp = *a;
  *a = *b;
  *b = tmp;
}
void bubbleSort(Point_t *pts, int len) {
  bool swapped = true;
  while (swapped) {
    int ii = 1;
    swapped = false;
    while (ii < len) {
      Point_t * prev = &pts[ii - 1];
      Point_t * curr = &pts[ii];
      if (less(*prev, *curr)) {
        swap(prev, curr);
        swapped = true;
      }
      ii++;
    }
  }
}
int main(void) {
  lenA = 4;
  lenB = 4;
  A = malloc(lenA);
  A = malloc(lenA);
  A = {{0,0}, {0,1}, {1,1}, {1,0}};
  B = {{1,0}, {1,1}, {0,1}, {0,0}};
  bubbleSort(A, lenA);
  bubbleSort(B, lenB);
  return 0;
}
```

```
less = fun(a, b) -> (res)
  tmp0 = fless(a.x, b.x)
  tmp1 = feq(a.x, b.x)
  tmp2 = fless(a.y, b.y)
  tmp3 = and(tmp1, tmp2)
  res = or(tmp0, tmp3)
swap = fun(*a, *b) -> (void)
  tmp = *a
  *a = *b
  *b = tmp
  void = 0
bubbleSort = fun(*pts, len) -> (void)
  swapped = true;
  while (swapped) {
    ii = 1;
    swapped = false;
    while (ii < len) {
      ptsii_1 = add(pts, subtract(ii, 1));
      ptsii = add(pts, ii);
      prev = &ptsii_1;
      curr = &pts_ii;
      dprev = *prev;
      dcurr = *curr;
      if (less(dprev, dcurr)) {
        swap(prev, curr);
        swapped = true;
      }
      ii = iadd(ii, 1);
    }
  }
  void = 0
lenA = 4;
lenB = 4;
A = allocate(lenA);
A = allocate(lenA);
A = {{0,0}, {0,1}, {1,1}, {1,0}};
B = {{1,0}, {1,1}, {0,1}, {0,0}};
bubbleSort(A, lenA);
bubbleSort(B, lenB);
return(0);
```

Steensgaard's Language

- No aggregate types
- Function variables are unique to the function
- Struct accessors flattened
- Pointer operations desugared to function calls
- Malloc converted to allocate
- Void return converted to use a dummy variable

```
less = fun(a, b) -> (res)
  tmp0 = fless(a, b)
  tmp1 = feq(a, b)
  tmp2 = fless(a, b)
  tmp3 = and(tmp1, tmp2)
  res = or(tmp0, tmp3)
swap = fun(*a, *b) -> (void)
  tmp = *a
  *a = *b
  *b = tmp
  void = 0
bubbleSort = fun(*pts, len) -> (void)
  swapped = true;
  while (swapped) {
    ii = 1;
    swapped = false;
    while (ii < len) {
      ptsii_1 = add(pts, subtract(ii, 1));
      ptsii = add(pts, ii);
      prev = &ptsii_1;
      curr = &pts_ii;
      dprev = *prev;
      dcurr = *curr;
      if (less(dprev, dcurr)) {
        swap(prev, curr);
        swapped = true;
      }
      ii = iadd(ii, 1);
    }
  }
  void = 0
lenA = 4;
lenB = 4;
A = allocate(lenA);
A = allocate(lenA);
A = {{0,0}, {0,1}, {1,1}, {1,0}};
B = {{1,0}, {1,1}, {0,1}, {0,0}};
bubbleSort(A, lenA);
bubbleSort(B, lenB);
return(0);
```

Algorithm

Steensgaard's Algorithm

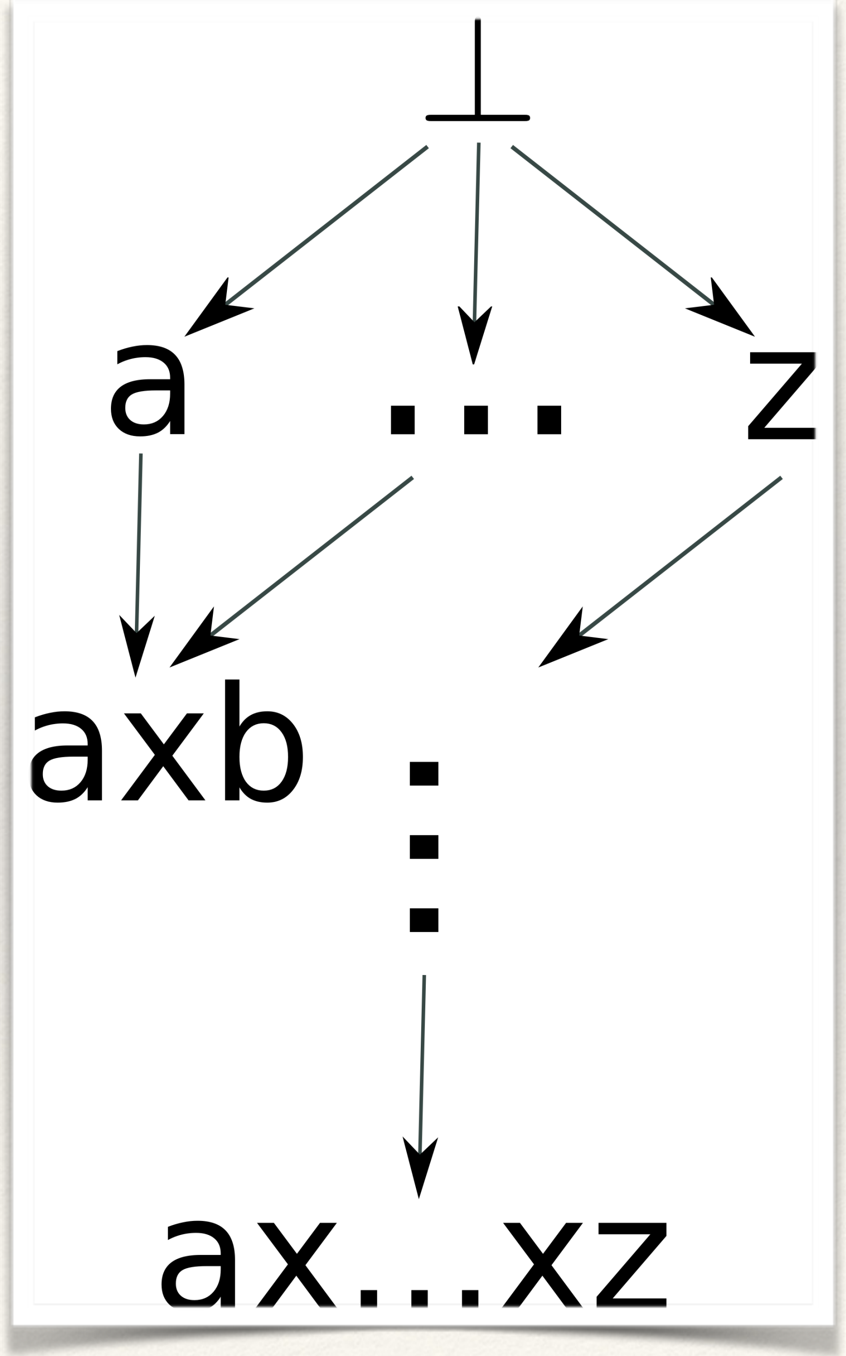
- Flow insensitive, interprocedural, context insensitive unification based pointer analysis
- Performs pointer analysis in near linear time
- Traverse the instructions from top to bottom in a single pass
- If type information is known then unify them, otherwise delay the unification

Modeling Values

- Model values as pointers to locations or points to functions
  - **Value types**: $\boldsymbol{\alpha} = \boldsymbol{\tau} \times \boldsymbol{\lambda}$
  - **Pointer/Address types**: $\boldsymbol{\tau} = \mathrm{ref}(\boldsymbol{\alpha}) \mid \perp$
  - **Function signature**: $\lambda = \mathrm{lam}(\boldsymbol{\alpha}_1,\ldots)(\boldsymbol{\alpha}_k,\ldots) \mid \perp$
- 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 ref($\perp\times\perp$)



Use a Simpler Example

```
typedef struct {
	float x, y;
} Point_t;
bool lessX(Point_t * a, Point_t * b) {
	return a->x < b->x;
}
...
Point_t A[1] = {{0, 0}};
Point_t B = {0, 1};
Point_t * pC;
bool g = lessX(&A[0], &B);
if (g) {
	pC = &A[0];
} else {
	pC = &B;
}
bool k = lessX(&B, &A[0]);
```

```
lessX = fun(lessa, lessb) -> (res)
	res = fpless(lessa, lessb)
v00 = 00
v01 = 01
A = allocate(8)
*A = v00
B = v01
A0 = *A
pA0 = &A0
pB = &B
g = lessX(pA0, pB)
if g then
	pC = &A0
else
	pC = &B
end
k = lessX(pB, pA0)
```

Typing Program

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

```
lessX = fun(lessa, lessb) -> (res)
    res = fpless(lessa, lessb)
```

$$\text{fpless} : \tau_0 = \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1, \alpha_2)(\alpha_3))$$

$$\text{lessa} : \tau_1 = \mathbf{ref}(\alpha_4)$$

$$\text{lessb} : \tau_2 = \mathbf{ref}(\alpha_5)$$

$$\text{res} : \tau_3 = \mathbf{ref}(\alpha_6) = \mathbf{ref}(\bot \times \bot)$$

One can use C's type information to know that the return type is not a pointer.

$$\text{less} : \tau_4 = \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_4, \alpha_5)(\alpha_6))$$

$$\alpha_4 \trianglelefteq \alpha_1$$

$$\alpha_5 \trianglelefteq \alpha_2$$

Constrains on the unbound types.

$$\alpha_6 \trianglelefteq \alpha_3$$

Typing Program

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

```
v00 = 00
v01 = 01
A = allocate(8)
*A = v00
```

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

$$\mathrm{v00} : \tau_5 = \mathbf{ref}(\bot \times \bot) = \mathbf{ref}(\alpha_7)$$

$$\mathrm{v01} : \tau_6 = \mathbf{ref}(\bot \times \bot) = \mathbf{ref}(\alpha_8)$$

Values are assumed to not reference anything.

$$\mathrm{A} : \tau_{10} = \mathbf{ref}(\mathbf{ref}(\_) \times \_)$$

Underscore matches any symbol

$$\mathrm{A} : \tau_{10} = \mathbf{ref}(\mathbf{ref}(\alpha_9) \times \_)$$

$$\alpha_7 \trianglelefteq \alpha 9$$

Constraints are added in case values are no longer ⊥

Typing Program

```
B = v01
A0 = *A
pA0 = &A0
pB = &B
```

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

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

$$\mathrm{B} : \tau_{11} = \mathbf{ref}(\alpha_{10})$$

$$\alpha_8 \trianglelefteq \alpha 10$$

$$\mathrm{A0} : \tau_{12} = \mathbf{ref}(\alpha_{11})$$

$$\alpha_9 \trianglelefteq \alpha 11$$

$$\mathrm{pA0} : \tau_{13} = \mathbf{ref}(\tau_{12} \times \_)$$

$$\mathrm{pB} : \tau_{14} = \mathbf{ref}(\tau_{11} \times \_)$$

Typing Program

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

```
g = lessX(pA0, pB)
```

$$\tau_{13} \trianglelefteq \alpha_4$$

$$\tau_{14} \trianglelefteq \alpha_5$$

Constraints from the function call.

$$\mathrm{g} : \tau_{15} = \mathbf{ref}(\bot \times \bot)$$

Typing Program

```
if g then
        pC = &A0
else
        pC = &B
end
```

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

$$\text{pC} : \tau_{16} = \mathbf{ref}(\tau_{12} \times \_)$$

$$\text{pC} : \tau_{16} = \mathbf{ref}(\tau_{11} \times \_)$$

$$\tau_{12} = \tau_{11}$$

Equality follows from the unifying the types

$$\alpha_{10} = \alpha_{11}$$

Typing Program

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

```
k = lessX(pB, pA0)
```

$$\tau_{14} \trianglelefteq \alpha_4$$

$$\tau_{13} \trianglelefteq \alpha_5$$

Why is this important?

$$g : \tau_{15} = \mathbf{ref}(\perp \times \perp)$$

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

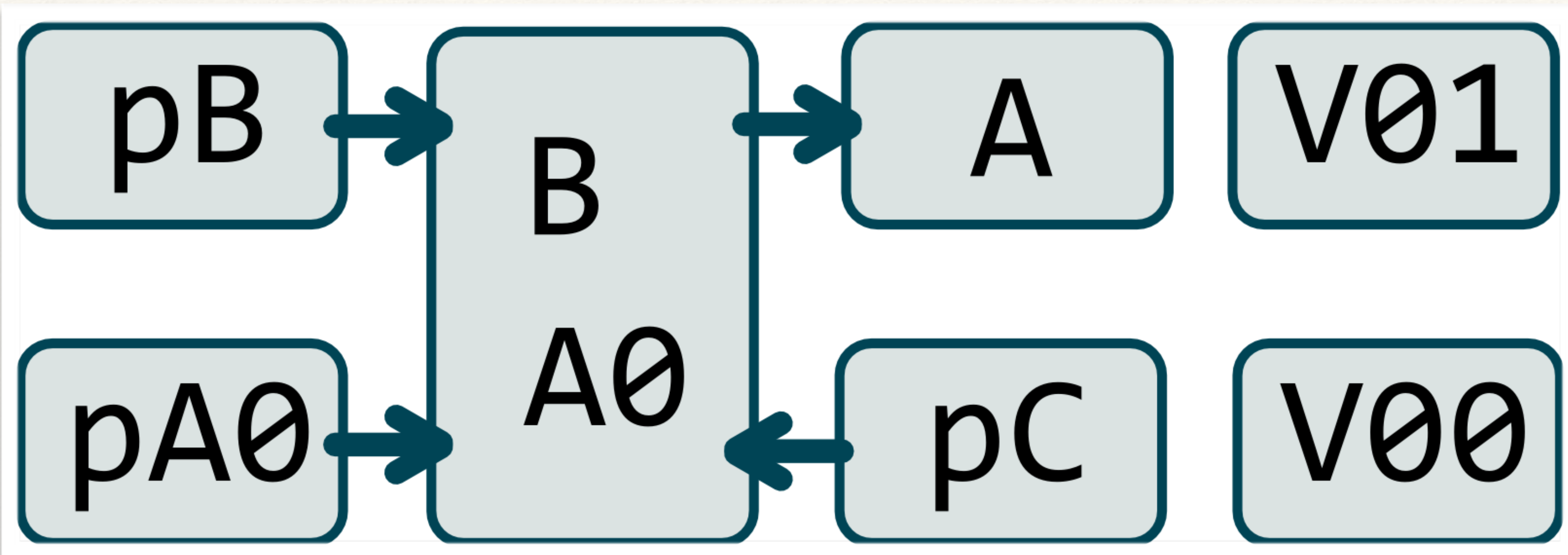
```
memcpy(A, B, sz);
memcpy(B, C, sz);
memset(B, 0, sz);
strcmp(s1, s2);
strcmp(s2, s3);
```

Final Contraint Set

$$\text{fpless}: \tau_0 = \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1, \alpha_2)(\alpha_3))$$
$$\text{lessa}: \tau_1 = \mathbf{ref}(\alpha_4)$$
$$\text{lessb}: \tau_2 = \mathbf{ref}(\alpha_5)$$
$$\text{res}: \tau_3 = \mathbf{ref}(\alpha_6) = \mathbf{ref}(\perp \times \perp)$$
$$\text{less}: \tau_4 = \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_4, \alpha_5)(\alpha_6))$$
$$\alpha_4 \trianglelefteq \alpha_1$$
$$\alpha_5 \trianglelefteq \alpha_2$$
$$\alpha_6 \trianglelefteq \alpha_3$$
$$\text{v00}: \tau_5 = \mathbf{ref}(\perp \times \perp) = \mathbf{ref}(\alpha_7)$$
$$\text{v01}: \tau_6 = \mathbf{ref}(\perp \times \perp) = \mathbf{ref}(\alpha_8)$$
$$\text{A}: \tau_{10} = \mathbf{ref}(\mathbf{ref}(\_) \times \_)$$
$$\text{A}: \tau_{10} = \mathbf{ref}(\mathbf{ref}(\alpha_9) \times \_)$$
$$\alpha_7 \trianglelefteq \alpha 9$$
$$\text{B}: \tau_{11} = \mathbf{ref}(\alpha_{10})$$
$$\alpha_8 \trianglelefteq \alpha 10$$

$$\text{A0}: \tau_{12} = \mathbf{ref}(\alpha_{11})$$
$$\alpha_9 \trianglelefteq \alpha 11$$
$$\text{pA0}: \tau_{13} = \mathbf{ref}(\tau_{12} \times \_)$$
$$\text{pB}: \tau_{14} = \mathbf{ref}(\tau_{11} \times \_)$$
$$\tau_{13} \trianglelefteq \alpha_4$$
$$\tau_{14} \trianglelefteq \alpha_5$$
$$\text{g}: \tau_{15} = \mathbf{ref}(\perp \times \perp)$$
$$\text{pC}: \tau_{16} = \mathbf{ref}(\tau_{12} \times \_)$$
$$\text{pC}: \tau_{16} = \mathbf{ref}(\tau_{11} \times \_)$$
$$\tau_{12} = \tau_{11}$$
$$\alpha_{10} = \alpha_{11}$$
$$\tau_{14} \trianglelefteq \alpha_4$$
$$\tau_{13} \trianglelefteq \alpha_5$$
$$\text{g}: \tau_{15} = \mathbf{ref}(\perp \times \perp)$$

The Point-to Set for Our Program




```
lessX = fun(lessa, lessb) -> (res)
      res = fpless(lessa, lessb)
v00 = 00
v01 = 01
A = allocate(8)
*A = v00
B = v01
A0 = *A
pA0 = &A0
pB = &B
g = lessX(pA0, pB)
if g then
      pC = &A0
else
      pC = &B
end
k = lessX(pB, pA0)
```

Why Partial Ordering Matters

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

```
v00 = 00
v01 = 01
A = allocate(8)
*A = v00
```

*Would incorrectly alias **v00** and **v01***

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

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 $\mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_1, \lambda_2)$

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

x = ∗y:
let $\mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}))$ in
if $\mathbf{type}(\tau_2) = \bot$ then
$\mathbf{settype}(\tau_2, \mathbf{ref}(\tau_1 \times \lambda_1))$
else
let $\mathbf{ref}(\tau_3 \times \lambda_3) = \mathbf{type}(\tau_2)$ in
if $\tau_1 \neq \tau_3$ then $\mathbf{cjoin}(\tau_1, \tau_3)$
if $\lambda_1 \neq \lambda_3$ then $\mathbf{cjoin}(\lambda_1, \lambda_3)$

x = op($\mathsf{y}_1 \ldots \mathsf{y}_n$):
for $i \in [1 \ldots n]$ do
let $\mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_1, \lambda_2)$

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

∗x = y:
let $\mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}))$
if $\mathbf{type}(\tau_1) = \bot$ then
$\mathbf{settype}(\tau_1, \mathbf{ref}(\tau_2 \times \lambda_2))$
else
let $\mathbf{ref}(\tau_3 \times \lambda_3) = \mathbf{type}(\tau_1)$ in
if $\tau_2 \neq \tau_3$ then $\mathbf{cjoin}(\tau_3, \tau_2)$
if $\lambda_2 \neq \lambda_3$ then $\mathbf{cjoin}(\lambda_3, \lambda_2)$

x = fun($\mathsf{f}_1 \ldots \mathsf{f}_n$)→($\mathsf{r}_1 \ldots \mathsf{r}_m$) S^*:
let $\mathbf{ref}(\_ \times \lambda) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
if $\mathbf{type}(\lambda) = \bot$ then
$\mathbf{settype}(\lambda, \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))$
where
$\mathbf{ref}(\alpha_i) = \mathbf{type}(\mathbf{ecr}(\mathsf{f}_i))$, for $i \leq n$
$\mathbf{ref}(\alpha_i) = \mathbf{type}(\mathbf{ecr}(\mathsf{r}_{i-n}))$, for $i > n$
else
let $\mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathbf{type}(\lambda)$ in
for $i \in [1 \ldots n]$ do
let $\tau_1 \times \lambda_1 = \alpha_i$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{f}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_2, \tau_1)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_2, \lambda_1)$
for $i \in [1 \ldots m]$ do
let $\tau_1 \times \lambda_1 = \alpha_{n+i}$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{r}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_1, \lambda_2)$

$\mathsf{x}_1 \ldots \mathsf{x}_m$ = p($\mathsf{y}_1 \ldots \mathsf{y}_n$):
let $\mathbf{ref}(\_ \times \lambda) = \mathbf{type}(\mathbf{ecr}(\mathsf{p}))$ in
if $\mathbf{type}(\lambda) = \bot$ then
$\mathbf{settype}(\lambda, \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))$
where
$\alpha_i = \tau_i \times \lambda_i$
$[\tau_i, \lambda_i] = \mathbf{MakeECR}(2)$
let $\mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathbf{type}(\lambda)$ in
for $i \in [1 \ldots n]$ do
let $\tau_1 \times \lambda_1 = \alpha_i$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_1, \lambda_2)$
for $i \in [1 \ldots m]$ do
let $\tau_1 \times \lambda_1 = \alpha_{n+i}$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_2, \tau_1)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_2, \lambda_1)$

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

cjoin(e_1, e_2):
  if **type**$(e_2) = \perp$ 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 $t_1 = \perp$ then
    type$(e) \leftarrow t_2$
    if $t_2 = \perp$ 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 $t_2 = \perp$ 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.

$$\begin{tikzcd} & y \\ p \arrow[ur] & \\ & x \end{tikzcd}$$

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

$$\begin{tikzcd} & y \\ p \arrow[ur] \arrow[dr] & \\ & x \end{tikzcd}$$

```
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;
```

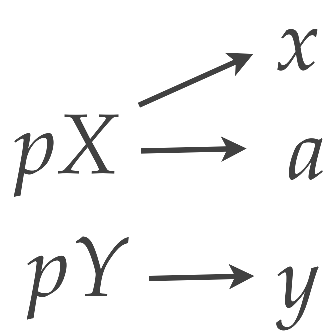
Flow Insensitive: Pointer analysis ignores the order of statements in the program.

$$p1 \longrightarrow y$$

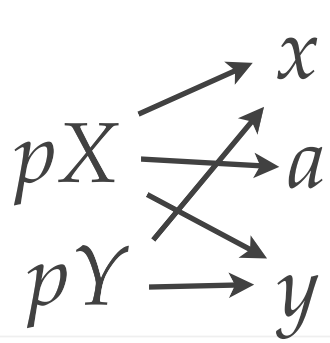
$$p0 \longrightarrow x$$

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