

Bjarne Steensgaard

Points-to Analysis in Almost Linear Time

Presented by **Abdul Dakkak**

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 p and q are pointers then $*p === *q$)
- ❖ A *points-to set* for p are the set of variables which p may point to

Overview

- ❖ Definitions
- ❖ Steensgaard's Language
- ❖ Steensgaard's Algorithm
- ❖ Results
- ❖ Conclusion

Definitions

- ❖ An analysis is *flow sensitive* if it computes accounting for the branches in the control flow graph.
- ❖ An analysis is *context sensitive* if keeps flow along different call paths separate.
- ❖ An analysis is *field sensitive* if it distinguishes between elements in a field (**struct** in C)

Definitions

Unification

- ❖ A unification algorithm finds a set of replacement rules such 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 equality

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;
```

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;
}
```


Steensgaard's Language

Translating into Steensgaard's Language

- ❖ Models C language
- ❖ C pointer operations can be desugared

$S ::= x = y$
 $\quad | x = \&y$
 $\quad | x = *y$
 $\quad | *x = y$
 $\quad | x = \text{op}(y \dots)$
 $\quad | x = \text{allocate}(y)$
 $\quad | x = \text{fun}(a \dots) \rightarrow (r \dots) S^*$
 $\quad | x \dots = p(a \dots)$

```

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 = &ptsii;
            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);

```

Bubble Sort

- ❖ 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 = &ptsii;
      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

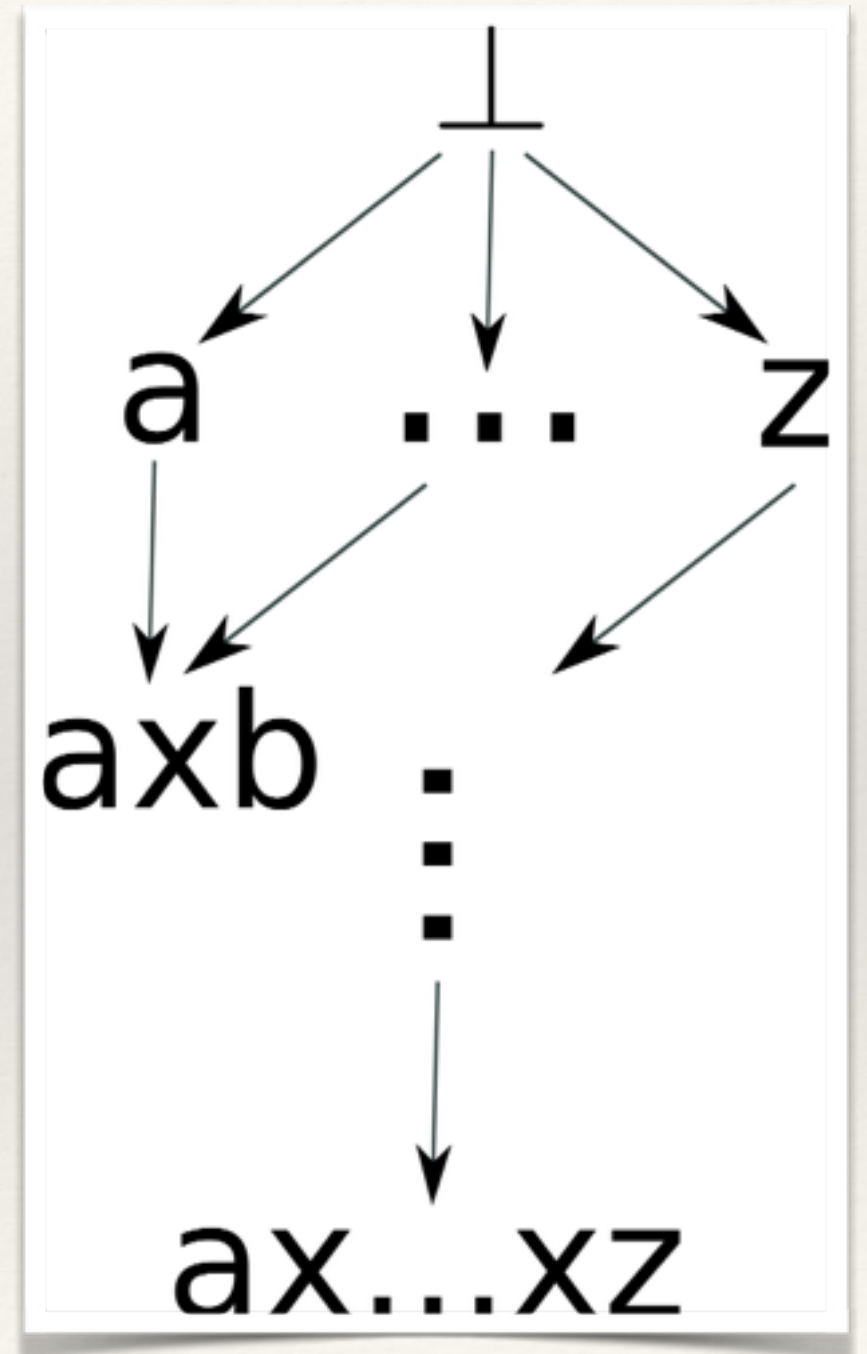
- ❖ Flow insensitive, context insensitive unification based pointer analysis
- ❖ Performs pointer analysis in near linear time
- ❖ Traverse the instructions from top to bottom in a single pass

Modeling Values

- ❖ Model values as pointers to locations or points to functions
 - ❖ Value types: $\alpha = \tau \times \lambda$
 - ❖ Pointer/Address types: $\tau = \text{ref}(\alpha) \mid \perp$
 - ❖ Function signature: $\lambda = \text{lam}(\alpha_1, \dots)(\alpha_k, \dots) \mid \perp$
- ❖ Impose partial order on memory locations

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 type of each type variable is initially assumed to be $\text{ref}(\perp \times \perp)$



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


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

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

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

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

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

$\mathit{res} : \tau_3 = \mathbf{ref}(\alpha_6) = \mathbf{ref}(\perp \times \perp)$ 

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

$\alpha_4 \sqtriangleleft \alpha_1$

$\alpha_5 \sqtriangleleft \alpha_2$

$\alpha_6 \sqtriangleleft \alpha_3$



Constrains on the unbound types.

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

Typing Program

v00 = 00

v01 = 01

A = allocate(8)

*A = v00

$v00 : \tau_5 = \mathbf{ref}(\perp \times \perp)$

$v01 : \tau_6 = \mathbf{ref}(\perp \times \perp)$ ←

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

$A : \tau_{10} = \mathbf{ref}(\tau_5 \times _)$ ←

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

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

Values are assumed to not reference anything.

Underscore matches any symbol

Typing Program

$B = v01$

$A0 = *A$

$pA0 = \&A0$

$pB = \&B$

$B : \tau_{11} = \tau_6$

$A0 : \tau_{12} = \tau_5$

$pA0 : \tau_{13} = \mathbf{ref}(\tau_5 \times _)$

$pB : \tau_{14} = \mathbf{ref}(\tau_6 \times _)$

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

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

Typing Program

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

`g = lessX(pA0, pB)`

$$\tau_5 \sqsubseteq \alpha_4$$

$$\tau_6 \sqsubseteq \alpha_5$$

Constraints from the function call.


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

Typing Program

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

$pC : \tau_{16} = \mathbf{ref}(\tau_5 \times _)$

$pC : \tau_{16} = \mathbf{ref}(\tau_6 \times _)$

$\tau_5 = \tau_6$ 

Equality follows from the above 2 constraints

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

Typing Program

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

k = lessX(pB, pA0)

$$\tau_6 = \alpha_4$$

$$\tau_5 = \alpha_5$$

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



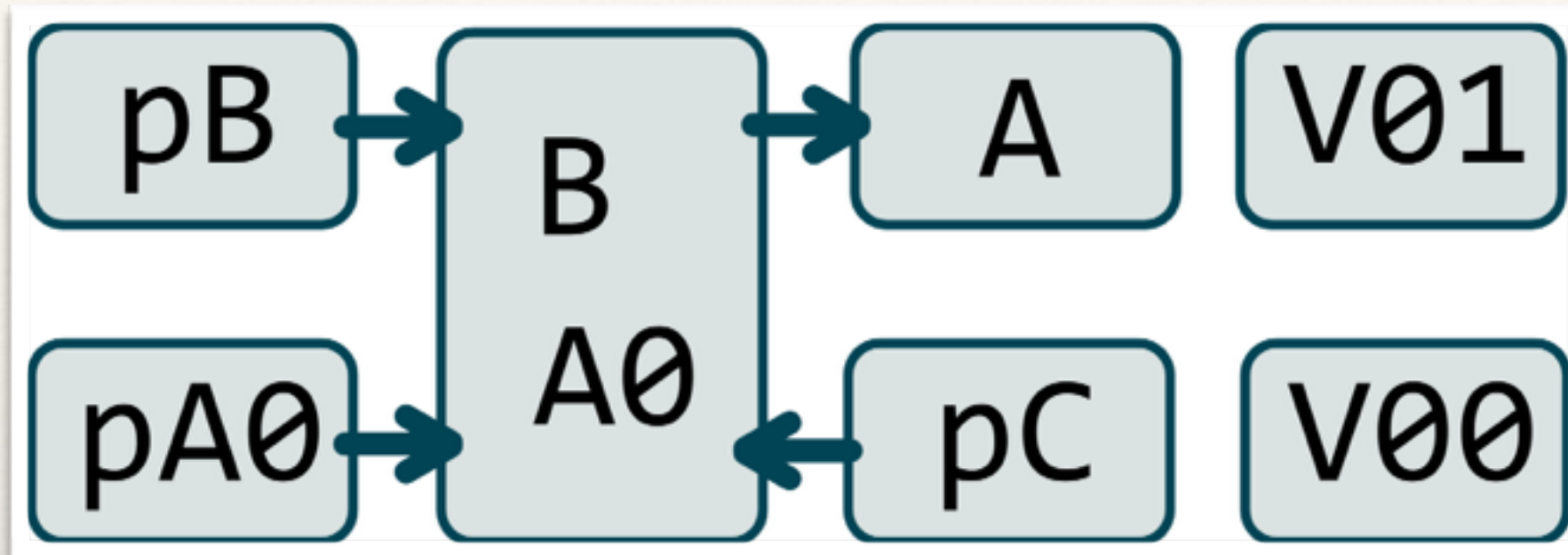
Why is this important?

Steensgaard Unifies Function Arguments

- ❖ If $t_1 \sqsubseteq C$ and $t_2 \sqsubseteq C$ then $t_1 = t_2$
 - ❖ Recall $(t \sqsubseteq C) \Leftrightarrow (t = \perp) \vee (t = C)$
- ❖ 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);
```


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

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

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 alias class contained 120 variables.

[illegible]

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	...	624																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																			
landi:allroots	0	1																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						
landi:assembler	10	8	3		3	1					1																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													

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 ... 624
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				1
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)

Conclusion

- ❖ First algorithm to scale beyond 10kLOC
- ❖ 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

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

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

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

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

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

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

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

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

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)) in
  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( $f_1 \dots f_n$ )  $\rightarrow$  ( $r_1 \dots r_m$ )  $S^e$ :
  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( $t_1, t_2$ )

```

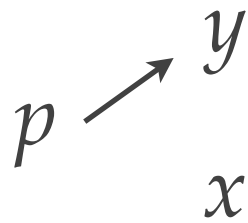
Unification

Union Find Data Structure



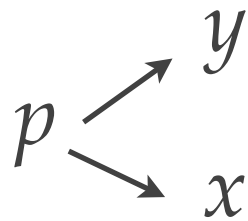
Flow Sensitive vs Insensitive Analysis

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



```
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.

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

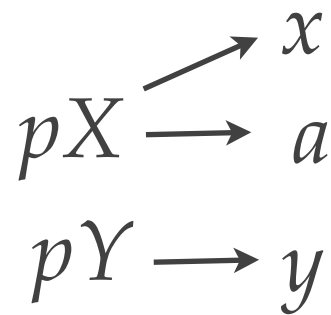
```
x = 1;  
y = 2;  
p0 = &x;  
p1 = &y;
```

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

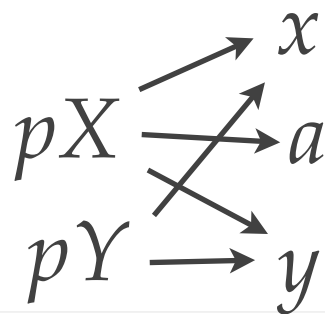
$$p1 \longrightarrow y$$
$$p0 \longrightarrow x$$

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

Field Sensitive Analysis



Object Sensitive Analysis



Heap Modeling





Point-to Analysis using Andersen



Point-to Analysis using BDD



Point-to Analysis using CFL Reachability



