Bjarne Steensgaard

Points-to Analysis in Almost Linear Time

Presented by Abdul Dakkak

Pointer Analysis Uses

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

```c
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...)
```

```c
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);
```

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

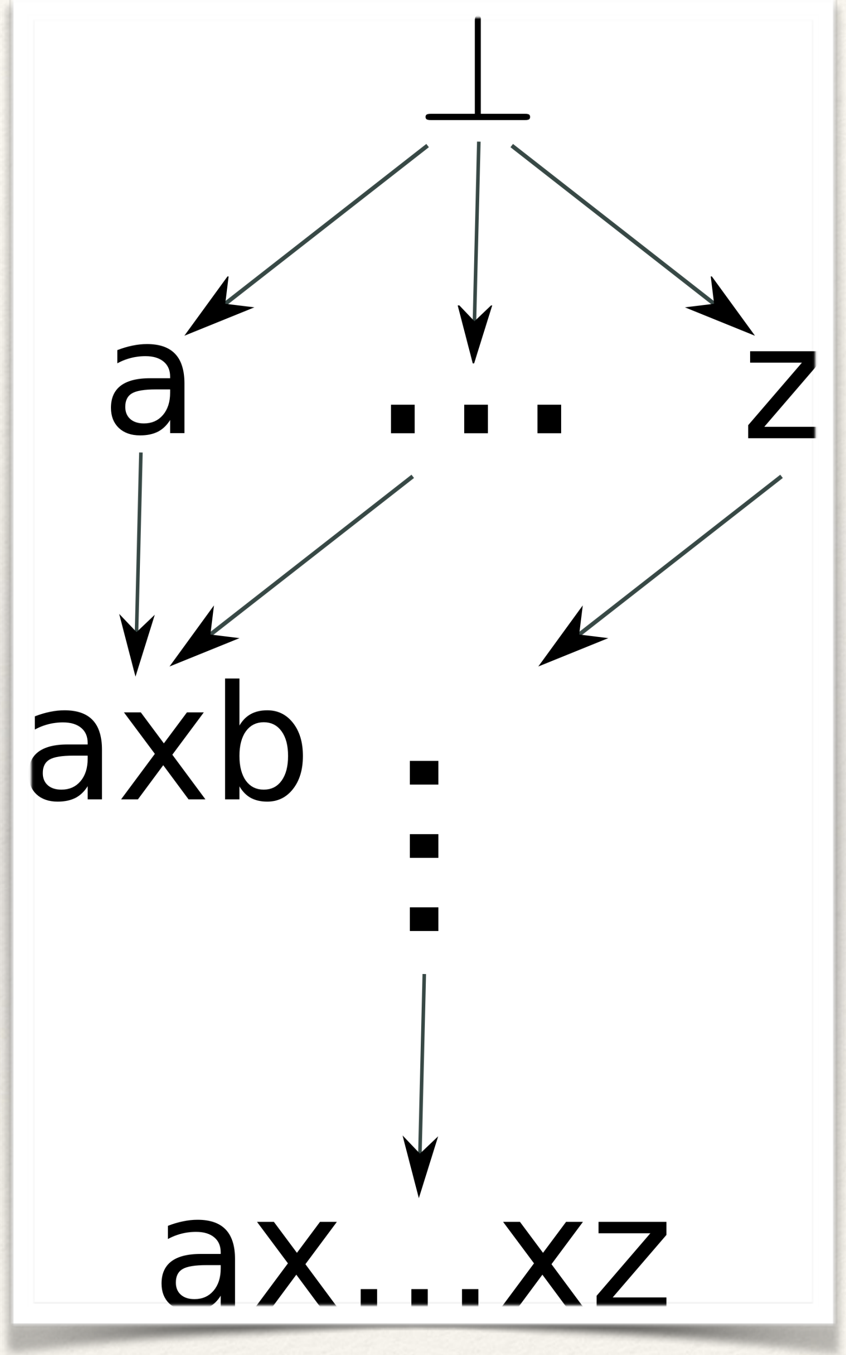
```
less = fun(a, b) -> (res)
  tmp0 = fless(a, b)
  tmp1 = feq(a, b)
  tmp2 = fless(a, b)
  tmp3 = and(tmp1, tmp2)
  res = or(tmp0, tmp3)
swap = fun(*a, *b) -> (void)
  tmp = *a
  *a = *b
  *b = tmp
  void = 0
bubbleSort = fun(*pts, len) -> (void)
  swapped = true;
  while (swapped) {
    ii = 1;
    swapped = false;
    while (ii < len) {
      ptsii_1 = add(pts, subtract(ii, 1));
      ptsii = add(pts, ii);
      prev = &ptsii_1;
      curr = &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 Outline

- Traverse the instructions from top to bottom
- Model values as pointers to locations or points to functions
  - $\boldsymbol{\alpha} = \boldsymbol{\tau} \times \boldsymbol{\lambda}$
  - $\boldsymbol{\tau} = \text{ref}(\boldsymbol{\alpha}) \mid \perp$
  - $\lambda = \text{lam}(\boldsymbol{\alpha}_1, \ldots)(\boldsymbol{\alpha}_k, \ldots) \mid \perp$
- 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)
```

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

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

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

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

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

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

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

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

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

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

Values are assumed to not reference anything.

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

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

Underscore matches any symbol

Typing Program

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

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

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

$$\begin{aligned}
\mathrm{B} :& \tau_{11} = \tau_6 \\
\mathrm{A0} :& \tau_{12} = \tau_5 \\
\mathrm{pA0} :& \tau_{13} = \mathbf{ref}(\tau_5 \times _) \\
\mathrm{pB} :& \tau_{14} = \mathbf{ref}(\tau_6 \times _)
\end{aligned}$$

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)
```

$$\tau_5 \trianglelefteq \alpha_4$$

Constraints from the function call.

$$\tau_6 \trianglelefteq \alpha_5$$

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

Typing Program

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

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

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

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

$$\tau_5 = \tau_6$$

Equality follows from the above 2 constraints

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)
```

$$\tau_6 = \alpha_4$$

$$\tau_5 = \alpha_5$$

Why is this important?

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

Steensgaard Unifies Function Arguments

- If $t_1 \trianglelefteq C$ and $t_2 \trianglelefteq C$ then $t_1 = t_2$
  - Recall $(t \trianglelefteq C) \Leftrightarrow (t = \bot) \vee (t = C)$
- 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

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

One alias 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
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												

# of vars.	0	1	2	3	4	5	6	. . .	30	31	32	33	. . .	44	45	. . .	52	. . .	74	. . .	78	. . .	83	. . .	113	. . .	120	. . .	285	. . .	613	. . .	624
landi:allroots																																	
landi:assembler												1															1						
landi:loader																																	
landi:compiler			1												1																		
landi:simulator			1	2													2						1										
landi:lex315				1																													
landi:football						1																											
austin:anagram																																	
austin:backprop																																	
austin:bc																					1												
austin:ft																																	
austin:ks																																	
austin:yacr2																																	
spec:compress																																	
spec:eqntott																																	
spec:espresso			1								1			1					1														
spec:li									1																								1
spec:sc					1																				1				1				
spec:alvinn																																	
spec:ear							1																										
LambdaMOO					1										1																1		

Author's Conclusions

Conclusion

- SSA provides some flow sensitivity for pointer analysis

Questions

Backup

Unification

- A unification algorithm finds a set of equations

Lorem Ipsum Dolor

Type Rules

Union Find Data Structure

Flow Sensitive Analysis

Context Sensitive Analysis

Field Sensitive Analysis

Object Sensitive Analysis

Heap Modeling

Point-to Analysis using Andersen

Point-to Analysis using BDD

Point-to Analysis using CFL Reachability