Points-To Analysis in Almost Linear Time

CSCI 3294

OCTOBER 9, 2001

Josh Bauman
Jason Bartkowiak

OUTLINE

Points To Analysis In Linear Time

- § **Paper Written by Bjarne Steensgaard(Microsoft Researcher)**
- § **Written in 1995 / Presented Jan. 1996**
- § **Flow Insensitive / Linear Time**
- § **Fastest Interprocedural Algorithm @ time of publication**
- § **Based on a NON-Standard Type System**

IMPORTANT ASPECTS OF THE RESEARCH

- **A Type System for describing a universally valid storage shape graph (SSG) for a program in Linear Space…**
- **A Constraint System which gives the algorithm better results…**
- **A Linear Time Algorithm for POINTS-TO Analysis by solving a constraint system…**

THE SOURCE LANGUAGE

THE SOURCE LANGUAGE

THE SYNTAX OF THE LANGUAGE IS AS FOLLOWS:

$$
\begin{array}{lll}
S & ::= & x = y \\
  & | & x = \&y \\
  & | & x = *y \\
  & | & x = op(y_1 \dots y_n) \\
  & | & x = allocate(y) \\
  & | & *x = y \\
  & | & x = fun(f_1 \dots f_n) \rightarrow (r_1 \dots r_m)\ S^* \\
  & | & x_1 \dots x_m = p(y_1 \dots y_n)
\end{array}
$$

Figure 1: Abstract syntax of the relevant statements, S, of the source language. x, y, f, r, and p range over the (unbounded) set of variable names and constants. op ranges over the set of primitive operator names. S^* denotes a sequence of statements. The control structures of the language are irrelevant for the purposes of this paper.

EXAMPLE OF FUNCTIONS IN THE LANGUAGE…..

```
Add1 = fun(x) ——→ ( r )
        xaddone = add(x 1)
        fi
result = Add1(99)
```

```
fact = fun(x)→(r)
  if lessthan(x 1) then
    r = 1
  else
    xminusone = subtract(x 1)
    nextfac = fact(xminusone)
    r = multiply(x nextfac)
  fi

result = fact(10)
```

TYPES

TYPES OF THE LANGUAGE

NON-STANDARD SET OF TYPES

•**Types describe:**

- **Location of variables and locations created by dynamic allocations**
- **A Set of Locations as well as possible runtime contents of those locations**

•**A TYPE can be viewed as a NODE in a SSG (Storage Shape Graph)**

The following productions describe our NONSTANDARD set of types used by our Points-To Analysis:

$$\alpha ::= \tau \times \lambda$$
$$\tau ::= \bot \mid \mathbf{ref}(\alpha)$$
$$\lambda ::= \bot \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})$$

TYPING RULES

TYPING RULES

•**Based on Non-Standard Set of Types**

•**Specify when a program is WELL-TYPED**

TYPING RULES (CONT'D.)

Before introducing the typing rules we must present the notion of partial ordering and why it is important to the language's typing rules…..

"obvious" typing rule

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

TYPING RULE (W/PARTIAL ORDERING)

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

Given that non-pointers are represented by type $\bot$, the requirement can now by expressed by the following typing rule:

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

TYPING RULES

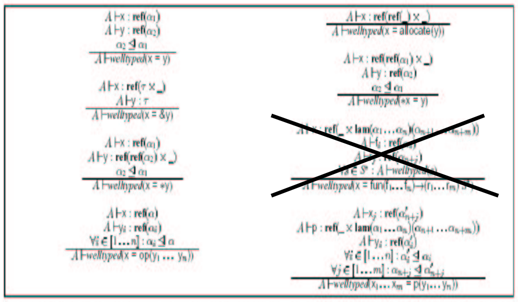


Figure 3: Type rules for the relevant statement types of the source language. All variables are assumed to have been associated with a type in the type environment $\mathcal{A}$. (Distinct variables are assumed to have distinct names, so the type environment can describe all variables in all scopes simultaneously.) "_" is a wild-card value in the rules, imposing no restrictions on the type component it represents.

EXPLANATION OF A STORAGE SHAPE GRAPH (SSG)

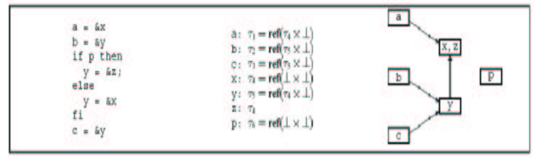


Figure 4: Example program, a typing of same that obeys the typing rules, and graphical representation of the corresponding storage shape graph. Note that variables **x** and **z** are described by the same type. Even though types τ_1 and τ_5 are structurally equivalent (as are τ_2 and τ_3, and τ_4 and τ_6), they are not considered the same types.

NOTE: TYPING SYSTEM ALLOWS ONLY ONE OUTGOING EDGE (TYPE VARIABLE CAN NOT BE ASSOCIATED WITH MORE THAN ONE TYPE)

ALGORITHM STAGES

ALGORITHM STAGES

- **Start w/ assumption that all variables are described by diff. types**
  - **Initially type of all variables is : $\mathbf{ref}(\perp \times \perp)$.**
  - **Type Variable consists of an equivalence class representative (ECR) with associated type info.**
- **Merge Types as necessary to ensure WELL-TYPEDNESS**
  - **Joining two types will NEVER make a statement that was well-typed no longer be well-typed…**
  - **AND…when all program statements are WELL-TYPED, the program is WELL-TYPED**

PROCESSING CONSTRAINTS

Type of Constraint : Inequality Constraint ($\trianglelefteq$)

If the "Left Hand Side" type variable is associated with type other than bottom, then two type variables MUST be joined to meet the constraint.

If "Left Hand Side" type variable is associated with "bottom" , then there is no need to join the two variables at this time…

INFERENCE RULES

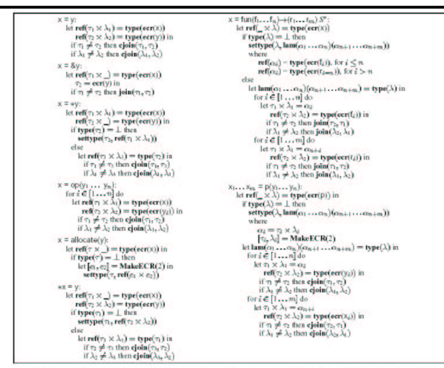
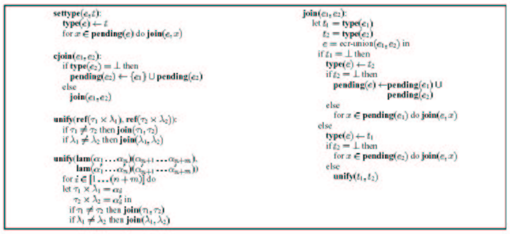


Figure 5: Inference rules corresponding to the typing rules given in Figure 3. **ecr**(x) is the ECR representing the type of variable x, and **type**(E) is the type associated with the ECR E. **cjoin**(x, y) performs the conditional join of ECRs x and y, and **settype**(E, X) associates ECR E with type X and forces the conditional joins with E. **MakeECR**(x) constructs a list of x new ECRs, each associated with the bottom type, $\perp$.

RULES FOR UNIFICATION OF TWO TYPES REPRESENTED BY ECR'S



```
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)
```

Figure 6: Rules for unification of two types represented by ECRs. We assume that ecr-union performs a (fast union/find) join operation on its ECR arguments and returns the value of a subsequent find operation on one of them.

COMPLEXITY

Space Cost = # of ECR's + # of JOIN operations

Initial # of ECR's – proportional to # of variables in the program

Time Cost - depends on "cost" of traversing statements of the program, "cost" of creating ECR's and types, the "cost" of performing JOIN operations, and the "cost" of find operations on ECR's

Implementation

- Written in Scheme
- Run time is linear

Implementation

- Table 1- All variables are included
- Table 2 - Some variables taken away
- Table 3 - Optimized form of Table 2

# 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	

Table 1

# 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																				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														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	

Table 2

# 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		

Table 3

Related Work

•Henglein - used type inference

•Weihl - points-to analysis is closest represented

Related Work

Flow-sensitive analyses

•Chase and Ruf’s algorithm - interprocedural data give polynomial time

•Emami - Exponential time

•Wilson and Lam - Exponential Time

Related Work

•Alias Analysis - Builds and maintains a list of access path expression that may evaluate to the same location.

•Context sensitive - Assumes runtime model that makes allocation regions explicit Related Work

Related Work

• Andersen –

•Non-sensitive = $O(A^2)$, A is the # of abstract locations.

•Sensitive = $O(A^4)$

•Compared to our solution, which is O(A)

Conclusion & Future

•Almost linear time

•Problems

•Future - Greater Precision

•Flow-sensitive

•Context-sensitive