

Lecture 11: Pointer Analysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

February 22, 2022

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Extending WHILE3ADDR with Pointers

I	$::=$...	
		$p := \&x$	taking the address of a variable
		$p := q$	copying a pointer from one variable to another
		$*p := q$	assigning through a pointer
		$p := *q$	dereferencing a pointer

Consider Constant Propagation

1 : $z := 1$

2 : $p := \&z$

3 : $*p := 2$

4 : **print** z

Need to know that line 3 changes variable z !

Consider Constant Propagation

1 : $z := 1$

2 : **if** (*cond*) $p := \&y$ **else** $p := \&z$

3 : $*p := 2$

4 : **print** z

Points-To Analysis: May vs. Must and Strong Updates

$$f_{CP}[\ast p := y](\sigma) =$$

$$f_{CP}[\ast p := y](\sigma) =$$

Points-To Analysis: May vs. Must and Strong Updates

$$f_{CP}[\ast p := y](\sigma) = \sigma[z \mapsto \sigma(y) \mid z \in \textit{must-point-to}(p)]$$

$$f_{CP}[\ast p := y](\sigma) = \sigma[z \mapsto \sigma(z) \sqcup \sigma(y) \mid z \in \textit{may-point-to}(p)]$$

Pointer Analysis

- Two common relations used as abstract values
 - Alias analysis: (x, y) alias pairs
 - Points-to analysis: $p \rightarrow q$ // or sets for points-to(p)
 - Both have *may* and *must* versions
- Very expensive to run precisely as data-flow analysis
 - Lattice is $2^{\text{Var} \times \text{Var}}$. Yikes!
 - Almost always needs to be inter-procedural
 - (even if used for intra-procedural optimizations)
 - Context-sensitivity is often important for adequate precision

Andersen's Analysis

- Flow-*insensitive* analysis
 - Considers only *nodes* of a CFG (i.e., instructions) and ignores all edges
 - What? Yes, really.
 - Trades-off precision for tractability
 - Can be combined with *context-sensitive* techniques
- Key idea: cast as constraint-solving problem
 - Abstract model of memory locations and points-to sets
 - Let l_x represent location of var x
 - Let p be the set of locations pointed-to by var p
 - One subset constraint per instruction
 - Invoke constraint solver. Done!

Andersen's Analysis

$$\overline{\llbracket p := \&x \rrbracket \hookrightarrow l_x \in p} \text{ address-of}$$

$$\overline{\llbracket p := q \rrbracket \hookrightarrow p \supseteq q} \text{ copy}$$

$$\overline{\llbracket *p := q \rrbracket \hookrightarrow *p \supseteq q} \text{ assign}$$

$$\overline{\llbracket p := *q \rrbracket \hookrightarrow p \supseteq *q} \text{ dereference}$$

Andersen's Analysis

$$\overline{\llbracket p := \&x \rrbracket \hookrightarrow l_x \in p} \text{ address-of}$$

$$\overline{\llbracket p := q \rrbracket \hookrightarrow p \supseteq q} \text{ copy}$$

$$\overline{\llbracket *p := q \rrbracket \hookrightarrow *p \supseteq q} \text{ assign}$$

$$\overline{\llbracket p := *q \rrbracket \hookrightarrow p \supseteq *q} \text{ dereference}$$

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \text{ copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \text{ assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \text{ dereference}$$

Example

```
x := 42
y := 108
q := &x
if (..)
  p := q
else
  p := &y
r = &p
s = *r
print(*s)
print(*q)
```

$$\frac{}{\llbracket p := \&x \rrbracket \hookrightarrow l_x \in p} \text{address-of}$$

$$\frac{}{\llbracket p := q \rrbracket \hookrightarrow p \supseteq q} \text{copy}$$

$$\frac{}{\llbracket *p := q \rrbracket \hookrightarrow *p \supseteq q} \text{assign}$$

$$\frac{}{\llbracket p := *q \rrbracket \hookrightarrow p \supseteq *q} \text{dereference}$$

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \text{copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \text{assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \text{dereference}$$

Dynamic Memory Allocation?

```
1 :  q := malloc()  
2 :  p := malloc()  
3 :  p := q  
4 :  r := &p  
5 :  s := malloc()  
6 :  *r := s  
7 :  t := &s  
8 :  u := *t
```

Dynamic Memory Allocation

```
1 :   $q := \text{malloc}()$   
2 :   $p := \text{malloc}()$   
3 :   $p := q$   
4 :   $r := \&p$   
5 :   $s := \text{malloc}()$   
6 :   $*r := s$   
7 :   $t := \&s$   
8 :   $u := *t$ 
```

$$\frac{}{\llbracket n: p := \text{malloc}() \rrbracket \hookrightarrow l_n \in p} \text{ malloc}$$

Exercise

```

1 :  q := malloc()
2 :  p := malloc()
3 :  p := q
4 :  r := &p
5 :  s := malloc()
6 :  *r := s
7 :  t := &s
8 :  u := *t

```

$$\frac{}{\llbracket p := \&x \rrbracket \hookrightarrow l_x \in p} \text{ address-of}$$

$$\frac{}{\llbracket p := q \rrbracket \hookrightarrow p \supseteq q} \text{ copy}$$

$$\frac{}{\llbracket *p := q \rrbracket \hookrightarrow *p \supseteq q} \text{ assign}$$

$$\frac{}{\llbracket p := *q \rrbracket \hookrightarrow p \supseteq *q} \text{ dereference}$$

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \text{ copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \text{ assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \text{ dereference}$$

$$\frac{}{\llbracket n: p := \text{malloc}() \rrbracket \hookrightarrow l_n \in p} \text{ malloc}$$

Efficiency

- $O(n)$ constraints
- $O(n)$ firings per copy-constraint
- $O(n^2)$ firings per assign/deref-constraint
- Worst-case $O(n^3)$ firings
- Can be solved in $O(n^3)$ time
 - McAllester [SAS'99]
- $O(n^2)$ in practice
 - Sridharan et al. [SAS'09]
 - K -sparseness property

$$\frac{}{\llbracket p := \&x \rrbracket \hookrightarrow l_x \in p} \text{ address-of}$$

$$\frac{}{\llbracket p := q \rrbracket \hookrightarrow p \supseteq q} \text{ copy}$$

$$\frac{}{\llbracket *p := q \rrbracket \hookrightarrow *p \supseteq q} \text{ assign}$$

$$\frac{}{\llbracket p := *q \rrbracket \hookrightarrow p \supseteq *q} \text{ dereference}$$

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \text{ copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \text{ assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \text{ dereference}$$

$$\frac{}{\llbracket n: p := \text{malloc}() \rrbracket \hookrightarrow l_n \in p} \text{ malloc}$$

Field-Sensitivity

1 : $p.f := \&x$

2 : $p.g := \&y$

Field-Sensitivity

1 : $p.f := \&x$

2 : $p.g := \&y$

A field-insensitive approach just treats fields `.f` as dereferences `*``.

Field-Sensitive Analysis

$$\overline{\llbracket p := q.f \rrbracket \hookrightarrow p \supseteq q.f} \text{ field-read}$$

$$\overline{\llbracket p.f := q \rrbracket \hookrightarrow p.f \supseteq q} \text{ field-assign}$$

Field-Sensitive Analysis

$$\overline{\llbracket p := q.f \rrbracket} \hookrightarrow p \supseteq q.f \text{ *field-read*}$$

$$\overline{\llbracket p.f := q \rrbracket} \hookrightarrow p.f \supseteq q \text{ *field-assign*}$$

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_q.f}{l_f \in p} \text{ *field-read*}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_p.f} \text{ *field-assign*}$$

Field-Sensitive Analysis

$$\overline{\llbracket p := q.f \rrbracket} \hookrightarrow p \supseteq q.f \text{ *field-read*}$$

$$\overline{\llbracket p.f := q \rrbracket} \hookrightarrow p.f \supseteq q \text{ *field-assign*}$$

```
a := new X()
b := new Y()
c := new Y()
a.f := b
a.g := c
print(a.f)
```

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_{q.f}}{l_f \in p} \text{ *field-read*}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_{p.f}} \text{ *field-assign*}$$

Steensgaard's Analysis

- **Problem:** Quadratic-in-practice is still not ultra-scalable
- **Challenge:** Need \sim LINEAR. How?
 - Solution space of pointer analysis (e.g. points-to sets) itself is $O(n^2)$.
- **Key idea:** Use constant-space per pointer. Merge aliases and alternates into the same equivalence class.
 - p can point to q or r ? Let's treat q and r as the same pseudo-var and merge everything we know about q and r .
 - Points-to "sets" are basically singletons

Steensgaard's Analysis - Example

1 : $p := \&x$

2 : $r := \&p$

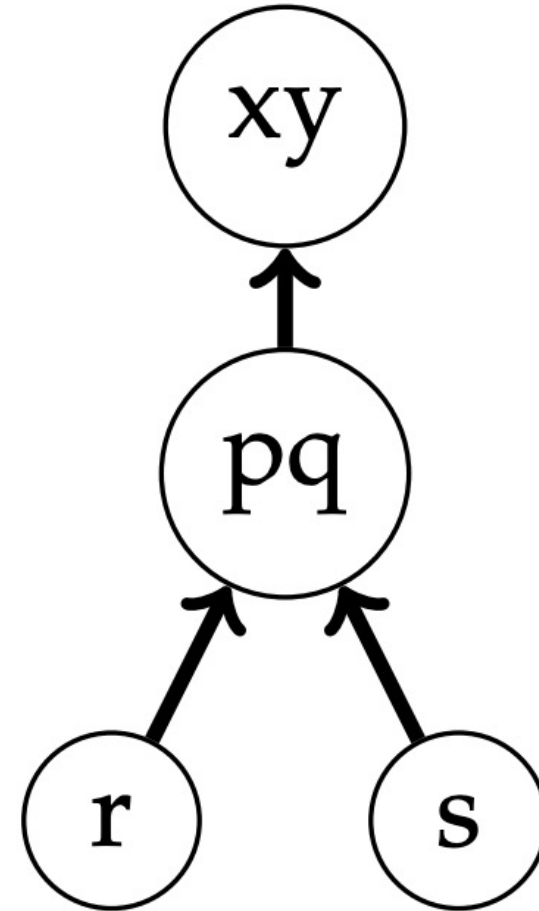
3 : $q := \&y$

4 : $s := \&q$

5 : $r := s$

Steensgaard's Analysis - Example

1 : $p := \&x$
2 : $r := \&p$
3 : $q := \&y$
4 : $s := \&q$
5 : $r := s$



Steensgaard's Analysis - **Exercise**

```
1 :   $a := \&x$   
2 :   $b := \&y$   
3 :  if  $p$  then  
4 :     $y := \&z$   
5 :  else  
6 :     $y := \&x$   
7 :   $c := \&y$ 
```


Steensgaard's Analysis

$$\overline{\llbracket p := q \rrbracket} \hookrightarrow \text{join}(*p, *q) \text{ } copy$$

$$\overline{\llbracket p := \&x \rrbracket} \hookrightarrow \text{join}(*p, x) \text{ } address-of$$

$$\overline{\llbracket p := *q \rrbracket} \hookrightarrow \text{join}(*p, **q) \text{ } dereference$$

$$\overline{\llbracket *p := q \rrbracket} \hookrightarrow \text{join}(**p, *q) \text{ } assign$$

Steensgaard's Analysis

$$\frac{}{\llbracket p := q \rrbracket \hookrightarrow \text{join}(*p, *q)} \text{copy}$$

$$\frac{}{\llbracket p := \&x \rrbracket \hookrightarrow \text{join}(*p, x)} \text{address-of}$$

$$\frac{}{\llbracket p := *q \rrbracket \hookrightarrow \text{join}(*p, **q)} \text{dereference}$$

$$\frac{}{\llbracket *p := q \rrbracket \hookrightarrow \text{join}(**p, *q)} \text{assign}$$

`join(ℓ_1, ℓ_2)`

`if (find(ℓ_1) == find(ℓ_2))
return`

`$n_1 \leftarrow * \ell_1$`

`$n_2 \leftarrow * \ell_2$`

`union(ℓ_1, ℓ_2)`

`join(n_1, n_2)`

Steensgaard's Analysis

- Abstract locations implemented as *union-find* data structure
 - Each union and find operation takes $O(\alpha(n))$ time each
 - Total algorithm running time is $O(n * \alpha(n))$ ~ almost linear
 - Space consumption is linear
- In practice: very scalable
 - Millions of LoC

OOP: Dynamic Dispatch

```
class A { A foo(A x) { return x; } }  
class B extends A { A foo(A x) { return new D(); } }  
class D extends A { A foo(A x) { return new A(); } }  
class C extends A { A foo(A x) { return this; } }  
  
// in main()  
A x = new A();  
while (...)  
    x = x.foo(new B()); // may call A.foo, B.foo, or D.foo  
A y = new C();  
y.foo(x);                // only calls C.foo
```