Lecture 11: Pointer Analysis

17-355/17-655/17-819: Program Analysis

Rohan Padhye and Jonathan Aldrich

March 11, 2021

* Course materials developed with Claire Le Goues

Extending WHILE3ADDR with Pointers

$$
\begin{array}{lcll}
I & ::= & \ldots & \\
 & \mid & p := \&x & \text{taking the address of a variable} \\
 & \mid & p := q & \text{copying a pointer from one variable to another} \\
 & \mid & *p := q & \text{assigning through a pointer} \\
 & \mid & p := *q & \text{dereferencing a pointer}
\end{array}
$$

Consider Constant Propagation

$$
\begin{aligned}
1: &\quad z := 1 \\
2: &\quad p := \&z \\
3: &\quad *p := 2 \\
4: &\quad \text{print } z
\end{aligned}
$$

Need to know that line 3 changes variable z!

Consider Constant Propagation

$$
\begin{array}{ll}
1: & z := 1 \\
2: & \textbf{if } (cond)\ p := \&y \textbf{ else } p := \&z \\
3: & *p := 2 \\
4: & \textbf{print } z
\end{array}
$$

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

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

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

Pointer Analysis

- Two common relations used as abstract values
  - Alias analysis: (x, y) alias pairs
  - Points-to analysis: p --> q // or sets for points-to(p)
  - Both have *may* and *must* versions
- Very expensive to run precisely as data-flow analysis
  - Lattice is $2^{Var \times Var}$. 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

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

Andersen's Analysis

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

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

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

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

Example

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

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

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

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

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

Dynamic Memory Allocation?

$$
\begin{aligned}
& q := malloc() \\
& p := malloc() \\
& p := q \\
& r := \&p \\
& s := malloc() \\
& *r := s \\
& t := \&s \\
& u := *t
\end{aligned}
$$

Dynamic Memory Allocation

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

$$\frac{}{[\![n\colon p := malloc()]\!] \hookrightarrow l_n \in p}\ malloc$$

Exercise

$$
\begin{array}{ll}
1: & q := \mathit{malloc}() \\
2: & p := \mathit{malloc}() \\
3: & p := q \\
4: & r := \&p \\
5: & s := \mathit{malloc}() \\
6: & *r := s \\
7: & t := \&s \\
8: & u := *t
\end{array}
$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

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

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

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

$$\frac{}{[\![n\!: p := \mathit{malloc}()]\!] \hookrightarrow l_n \in p}\ \textit{malloc}$$

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

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

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

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

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

$$\frac{}{[\![n\!: p := \mathit{malloc}()]\!] \hookrightarrow l_n \in p}\ \textit{malloc}$$

Field-Sensitivity

$$1: \quad p.f := \&x$$
$$2: \quad p.g := \&y$$

Field-Sensitivity

$$1: \quad p.f := \&x$$
$$2: \quad p.g := \&y$$

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

Field-Sensitive Analysis

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f}\ \textit{field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q}\ \textit{field-assign}$$

Field-Sensitive Analysis

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f} \textit{ field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q} \textit{ field-assign}$$

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

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

Field-Sensitive Analysis

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f}\ \textit{field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q}\ \textit{field-assign}$$

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

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

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

Steensgaard's Analysis

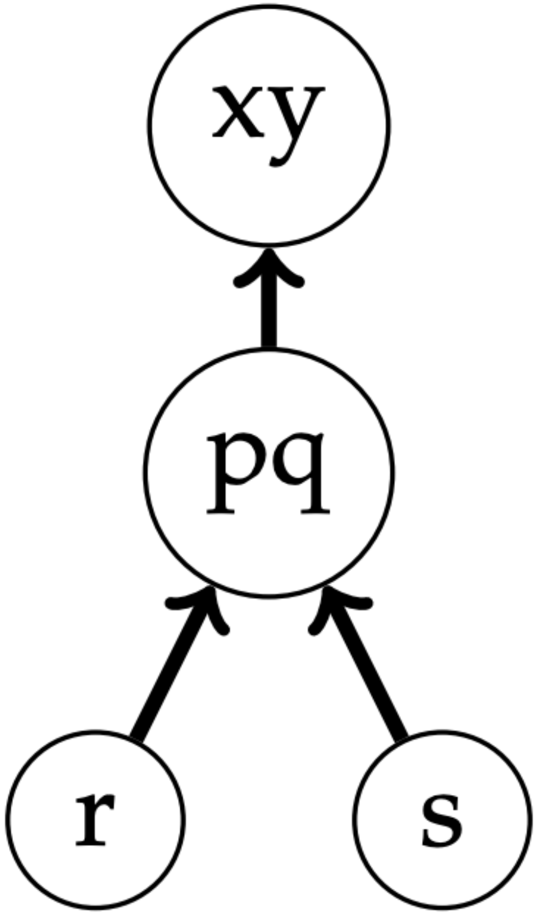
- **Problem**: Quadratic-in-practice is still not ultra-scalable
- **Challenge**: Need ~LINEAR. How?
  - Solution space of pointer analysis (e.g. points-to sets) itself is $O(n^2)$.
- **Key idea**: Use constant-space per pointer. Merge aliases and alternates into the same equivalence class.
  - *p* can 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

$$\begin{aligned}
1:&\quad p := \&x\\
2:&\quad r := \&p\\
3:&\quad q := \&y\\
4:&\quad s := \&q\\
5:&\quad r := s
\end{aligned}$$

Steensgaard's Analysis - Example

$$\begin{aligned} 1:&\quad p := \&x \\ 2:&\quad r := \&p \\ 3:&\quad q := \&y \\ 4:&\quad s := \&q \\ 5:&\quad r := s \end{aligned}$$



Steensgaard's Analysis - **Exercise**

```
a := &x
b := &y
if p then
    y := &z
else
    y := &x
c := &y
```

Steensgaard's Analysis

$$\frac{}{[\![p := q]\!] \hookrightarrow join(*p, *q)}\ copy$$

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

$$\frac{}{[\![p := *q]\!] \hookrightarrow join(*p, **q)}\ dereference$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow join(**p, *q)}\ assign$$

Steensgaard's Analysis

$$\frac{}{[\![p := q]\!] \hookrightarrow join(*p, *q)}\ copy$$

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

$$\frac{}{[\![p := *q]\!] \hookrightarrow join(*p, **q)}\ dereference$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow join(**p, *q)}\ assign$$

```
join(ℓ1, ℓ2)
    if (find(ℓ1) == find(ℓ2))
        return
    n1 ← *ℓ1
    n2 ← *ℓ2
    union(ℓ1, ℓ2)
    join(n1, n2)
```

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
```