Pointer/Alias Analysis

Lecture 7: CPEN 400P

Karthik Pattabiraman, UBC

(Based on Stephen Chong's lecture at Harvard Univ., CS252, and Vikram Adve's CS526 at the Univ. of Illinois)

Outline

What is pointer analysis ?

Types of pointer analysis

Anderson’s analysis

Steensgard analysis

What’s pointer analysis ?

Pointer analysis: What memory locations can a pointer expression refer to?

Alias analysis: When do two pointer expressions refer to the same location?

```
int x;

p = &x;

q = p;
```

What locations alias each other ?

*p and *q alias, as do x and *p, and x and *q

What causes Aliasing ?

Pointers, Pointer arithmetic etc.

- See previous slide

Call by reference

Array indexing, e.g., a[i], i = j; a[j] → a[i] and a[j] alias

Virtual functions in C++ (use of vtable)

Many other cases

What’s the use of pointer analysis ?

Useful in many analysis

- Live-out: Can lead to variables being killed

```
p = &x;
*p = a + b;
```

- Constant propagation: can lead to spurious constants

```
x = 3;
p = &x;
*p = 4;
```

Challenges of pointer analysis

1. Pointers to pointers, which can occur in many ways:
   - Take address of pointer
   - Pointer to structure containing pointer
   - Pass a pointer to a procedure by reference

2. Aggregate objects: structures and arrays containing pointers

3. Recursive data structures (lists, trees, graphs, etc.)

   closely related problem: anonymous heap locations

4. Control-flow: analyzing different data paths

5. Inter-procedural analysis is crucial

Common Simplifying Assumptions

1. Aggregate objects: arrays (and perhaps structures) containing pointers
   - Simple solution: treat as a single big object!
   - Pointer arithmetic is only legal for traversing an array:

     q = p ± i and q = &p[i] are handled the same as q = p

2. Recursive data structures (lists, trees, graphs, etc.)

   - Compute aliases, not “shape”
   - Use simplified naming schemes for heap objects (later slide)

3. Control-flow: analyzing different data paths

   - Flow-insensitive, path insensitive analysis

Outline

What is pointer analysis ?

Types of pointer analysis

Anderson’s analysis

Steensgard analysis

Flow-sensitivity

- Flow-sensitive analysis is one that computes a distinct result for each program point

- Flow-insensitive analysis generally computes a single result for an entire procedure or an entire program

- Flow-sensitive is too expensive in practice and rarely used
  - All the pointer analysis we'll consider in this class are **flow-insensitive**
  - SSA provides a limited form of flow-sensitivity within a procedure

Context Sensitivity

A context-sensitive interprocedural analysis computes results that may hold only for realizable paths through a program.

Needed for scaling up to 1000s of lines of code especially for security & reliability

Need to consider many parameters

- Heap cloning vs. no cloning: Cloning gives greater context-sensitivity
- Bottom-up vs. top-down: Does final result for a procedure include only “realizable” behavior from all contexts?
- Handling of recursive functions: Does analysis retain context-sensitivity within SCCs in the call graph?

Modeling memory locations example

Should we distinguish between calls to goo() ?

```
foo() {
        T* p = goo();
        T* q = goo();
}
goo(int n) {
        return new T(n);
}
```

What about calls to makeList() ?

```
T* makelist(int len) {
        T* newObj = new T;
        // Many distinct objects allocated here
        newObj->next = (--len == 0)?
                                NULL : makelist(len);
        return newObj;
}
```

How do we model memory locations ?

Global variables - Use a single node

Local variables - use a single node *per context*

Dynamically allocated memory - potentially *unbounded* locations at runtime

- One node for entire heap (too imprecise)
- One node for each type (requires type analysis
- One node per calling site (can lead to conflation across calling context)
- One node per calling context, i.e., for each allocation statement

“May” versus “Must” Analysis

May analysis: Aliasing that *may* occur during execution

Must analysis: Aliasing that *must* occur during execution

Consider liveness analysis: *p = *q + 4;

What’s the VarKill set for the statement ? What about the UEVar set ?

- May analysis: if *q may alias y, then y is in the UEVar set
- Must analysis: if *p must alias x, then x is in the VarKill set

Problem Statement

Flow-insensitive, **May analysis**

Assume program consists of only statements of the forms

- p = &a;
- p = q;
- *p = q;
- p = *q;

Assume pointers, q and address-taken variables are disjoint

We want to compute the points-to pairs, i.e., which pointer points to which variable

Outline

What is pointer analysis ?

Types of pointer analysis

Anderson’s analysis

Steensgard analysis

Andersen-style pointer analysis

- View pointer assignments as **subset constraints**
- Use constraints to propagate points-to information
- Takes O(N^3) time in the general case

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$
Complex	*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$

Andersen-style pointer analysis

- Can solve these constraints directly on sets pts(p)

```
p = &a;
q = p;
p = &b;
r = p;
```

$p \supseteq \{a\}$

$q \supseteq p$

$p \supseteq \{b\}$

$r \supseteq p$

$pts(p) = \{a, b\}$

$pts(q) = \{a, b\}$

$pts(r) = \{a, b\}$

$pts(a) = \varnothing$

$pts(b) = \varnothing$

Anderson’s Algorithm: Graph Representation

```
p = &a;

q = &b;

p = q;

r = &p;

*r = &c;

q = *r;
```

Anderson’s Algorithm: Graph Representation

p = &a; p → a

q = &b;

p = q;

r = &p;

*r = &c;

q = *r;

Anderson’s Algorithm: Graph Representation

p = &a;

q = &b;

p = q;

r = &p;

*r = &c;

q = *r;

p → a

q → b

Anderson’s Algorithm: Graph Representation

p = &a;

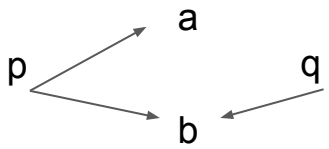
q = &b;

p = q;

r = &p;

*r = &c;

q = *r;



Anderson’s Algorithm: Graph Representation

p = &a;

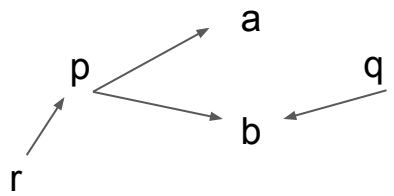
q = &b;

p = q;

r = &p;

*r = &c;

q = *r;



Anderson’s Algorithm: Graph Representation

p = &a;

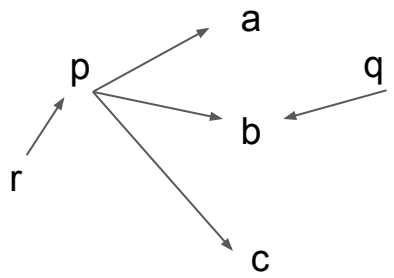
q = &b;

p = q;

r = &p;

***r = &c;**

q = *r;



Anderson’s Algorithm: Graph Representation

p = &a;

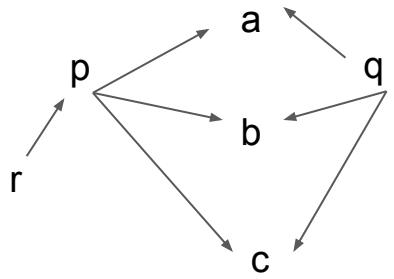
q = &b;

p = q;

r = &p;

*r = &c;

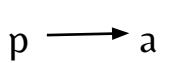
q = *r;



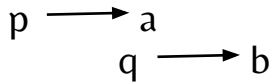
Points-to	Set
p	{a, b, c}
q	{a, b, c}
r	{p}
a	{}
b	{ }
c	{ }

Class Activity: Anderson's analysis

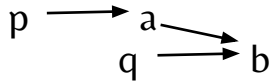
p = &a



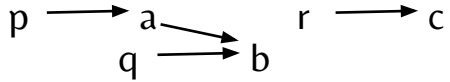
q = &b



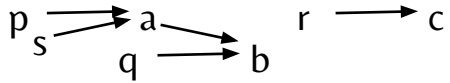
*p = q;



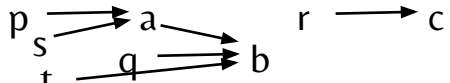
r = &c;



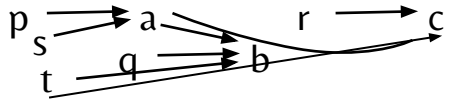
s = p;



t = *p;



*s = r;



Points-to	Set
p	{a}
q	{b}
r	{c}
s	{a}
t	{b, c}
a	{b, c}
b	{ }
c	{ }

Andersen-style as graph closure

- Can be cast as a graph closure problem
- One node for each pts(p), pts(a)

Assgmt.	Constraint	Meaning	Edge
a = &b	$a \supseteq \{b\}$	$b \in pts(a)$	no edge
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \rightarrow a$
a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$	no edge
*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$	no edge

- Each node has an associated points-to set
- Compute transitive closure of graph, and add edges according to complex constraints: O(N^3)

Anderson's algorithm: O(N^3)

Let W = { v | pts(v) ≠∅ } (all nodes with non-empty points to sets)

While W not empty

    v ← select from W

    for each a ∈ pts(v) do

        for each constraint p ⊇*v

            add edge a→ p, and add a to W if edge is new

        for each constraint *v ⊇ q

            add edge q→a, and add q to W if edge is new

    for each edge v→q do

        pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed

Outline

What is pointer analysis ?

Types of pointer analysis

Anderson’s analysis

Steensgard analysis

Steensgaard-style analysis

- Also a constraint-based analysis
- Uses **equality constraints** instead of subset constraints
- Less precise than Andersen-style, thus more scalable
  - Almost linear time compared to O(N^3)

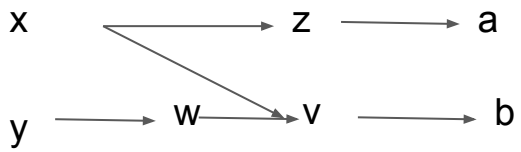
Constraint type	Assignment	Constraint	Meaning
Base	a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
Simple	a = b	a = b	pts(a) = pts(b)
Complex	a = *b	a = *b	∀ v∈pts(b). pts(a) = pts(v)
Complex	*a = b	*a = b	∀ v∈pts(a). pts(v) = pts(b)

Implementing Steensgaard-style analysis

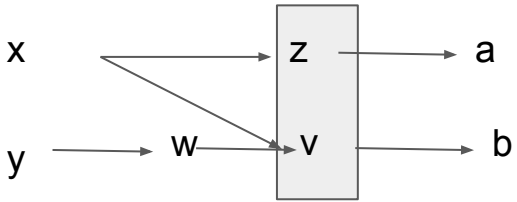
- Conceptually: restrict every node to only one outgoing edge
  - If $p \rightarrow x$ and $p \rightarrow y$, merge x and y ("Unify")
  - $\Rightarrow$ All objects "pointed to" by p are a single equivalence class

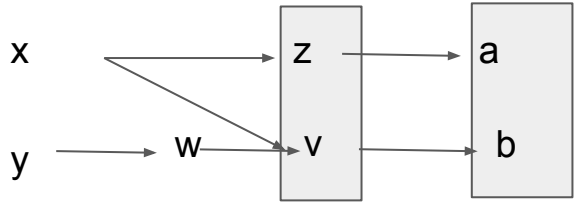
- Can be efficiently implemented using Union-Find algorithm
  - Nearly linear time: O(n): Tarjan's data-structure
  - Each statement needs to be processed just once

Example: Steensgard's algorithm



x = *y





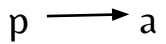
What's the precision ?

Class Activity

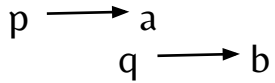
Can you analyze the same example we used for Anserson’s analysis with Steensgard-style analysis ?

Activity Solution (Original)

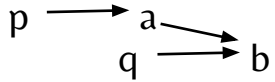
p = &a



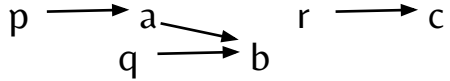
q = &b



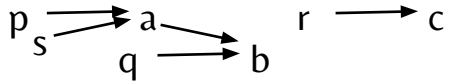
*p = q;



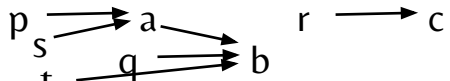
r = &c;



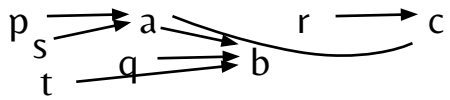
s = p;



t = *p;



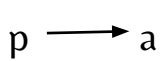
*s = r;



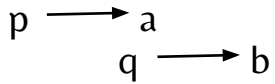
Points-to	Set
p	{a}
q	{b}
r	{c}
s	{a}
t	{b, c}
a	{b, c}
b	{ }
c	{ }

Activity Solution (Unification)

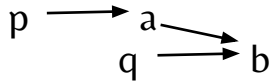
p = &a



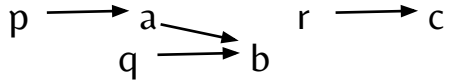
q = &b



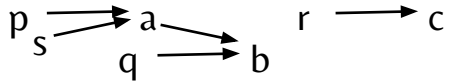
*p = q;



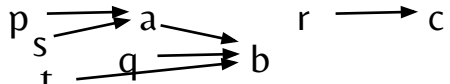
r = &c;



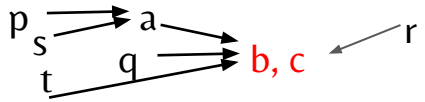
s = p;



t = *p;



*s = r;



Points-to	Set
p	{a}
q	{b, c}
r	{b, c}
s	{a}
t	{b, c}
a	{b, c}
b	{ }
c	{ }

Outline

What is pointer analysis ?

Types of pointer analysis

Anderson’s analysis

Steensgard analysis