Alias Analysis

Last time
- Alias analysis I (pointer analysis)
  - Address Taken
  - FIAlias, which is equivalent to Steensgaard

Today
- Alias analysis II (pointer analysis)
  - Anderson
  - Emami

Next time
- Midterm review

Properties of Alias Analysis

Scope: Intraprocedural (per procedure) or Interprocedural (whole program)

Representation
- Alias pairs - pairs of memory references that may access the same location
- Points-to sets - relations of the form (a->b) such that location a contains the address of location b
- Equivalence sets - all memory references in the same set may alias

Flow sensitivity: Sensitive versus insensitive

Context sensitivity: Sensitive versus insensitive

Definiteness: May versus must as well

Heap Modeling - How are dynamically allocated locations modeled?

Aggregate Modeling - are fields in structs or records modeled separately?
Address Taken

Algorithm overview
- Assume that nothing must alias
- Assume that all pointer dereferences may alias each other
- Assume that variables whose addresses are taken (and globals) may alias all pointer dereferences

Characterization of Address Taken
- Per procedure
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: equivalence sets
- Heap modeling: none
- Aggregate modeling: none

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

two equivalence sets
```
a
**a, *a, *b, *d, b, c, e, d
```

Steensgaard 96 equivalent to FIAlias [Ryder et. al. 2001]

Overview
- Uses unification constraints, for pointer assignments, p = q, Pts-to(p) = Pts-to(q).
  The union is done recursively for multiple-level pointers
- Almost linear in terms of program size, O(n)
- Uses fast union-find algorithm
- Imprecision stems from merging points-to sets

Characterization of Steensgaard
- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling: none
- Aggregate modeling: possibly

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

due to stmt 4

source: Barbara Ryder’s Reference Analysis slides
Unification Constraints

Conceptual Outline
- Add a constraint for each statement
- Solve the set of constraints

Steensgaard Constraints for C
- \( s: p = \&x; \quad x \in \text{Pts-to}(p) \)
- \( s: p = q; \quad \text{Pts-to}(p) = \text{Pts-to}(q) \)
- \( s: p = *q; \quad \forall a \in \text{Pts-to}(q), \text{Pts-to}(p) = \text{Pts-to}(a) \)
- \( s: *p = q; \quad \forall b \in \text{Pts-to}(p), \text{Pts-to}(b) = \text{Pts-to}(q) \)

Andersen 94

Overview
- Uses inclusion constraints, for pointer assignments, \( p = q, \text{Pts-to}(q) \subseteq \text{Pts-to}(p) \)
- Cubic complexity in program size, \( O(n^3) \)

Characterization of Andersen
- Whole program
- Flow-insensitive
- Context-insensitive
- May analysis
- Alias representation: points-to
- Heap modeling?
- Aggregate modeling: fields

source: Barbara Ryder’s Reference Analysis slides
Outline of Andersen’s Algorithm

Find all pointer assignments in the program

For each pointer assignment
   - For p ≡ q, all outgoing points-to edges from q are copied to be outgoing from p
   - If new outgoing edges are added to q during the algorithm they must also be copied to p

Using flow-insensitive, points-to
   - s: p = &x;
     x ∈ Pts-to(p)
   - s: p = q;
     Pts-to(q) ⊆ Pts-to(p)
   - s: p = *q;
     ∀a ∈ Pts-to(q), Pts-to(a) ⊆ Pts-to(p)
   - s: *p = q;
     ∀b ∈ Pts-to(p), Pts-to(q) ⊆ Pts-to(b)

Flow-sensitive May Points-To Analysis

Analogous flow functions
   - ⊓ is ∪
   - s: p = &x;
     out[s] = {(p→x)} ∪ (in[s] − {(p→y) ∀y})
   - s: p = q;
     out[s] = {(p→x) | (q→t) ∈ in[s]} ∪ (in[s] − {(p→y) ∀y})
   - s: p = *q;
     out[s] = {(p→x) | (q→t) ∈ in[s] & (x→t) ∈ in[s]} ∪ (in[s] − {(p→x) ∀x})
   - s: *p = q;
     out[s] = {(x→t) | (p→x) ∈ in[s] & (q→t) ∈ in[s]} ∪ (in[s] − {(x→x) ∀x | (p→x) ∈ in_{mix}[s]})
Flow-sensitive May Alias-Pairs Analysis

In the below data-flow equations, M and N represent any memory reference expression and + represents a specific number of dereferences. Meet function is $\cup$:

- s: $p = *x$
  
  $\text{out}[s] = \{(*p, x) \cup (\text{in}[s] - (*p, y) \forall y)\}$
  
  $\cup \{(*M, x) | (M, p) \in \text{in}[s]\} \cup \{(*+, M, N) | (M, p) \in \text{in}[s] \& (+x, N) \in \text{in}[s]\}$

- s: $p = q$
  
  $\text{out}[s] = \{(*p, t) | (*q, t) \in \text{in}[s]\} \cup (\text{in}[s] - (*p, y) \forall y)\}$
  
  $\cup \{(*M, t) | (M, p) \in \text{in}[s] \& (*q, t) \in \text{in}[s]\}$
  
  $\cup \{(*+, M, N) | (M, p) \in \text{in}[s] \& (*q, t) \in \text{in}[s] \& (+t, N) \in \text{in}[s]\}$

- s: $p = *q$
  
  $\text{out}[s] = \{(*p, r) | (*q, r) \in \text{in}[s] \& (*r, t) \in \text{in}[s]\} \cup \{(*M, r) | (M, p) \in \text{in}[s] \& (*q, r) \in \text{in}[s] \& (+t, N) \in \text{in}[s]\}$

Other Issues (Modeling the Heap)

Issue
- Each allocation creates a new piece of storage
  
  e.g., $p = \text{new } T$

Proposal?
- Generate (at compile-time) a new “variable” to stand for new storage
- $\text{newvar}$: Creates a new variable

Flow function
- s: $p = \text{new } T$
  
  $\text{out}[s] = \{(p, \text{newvar})\} \cup (\text{in}[s] - (p, x) \forall x)$

Problem
- Domain is unbounded!
- Iterative data-flow analysis may not converge
Modeling the Heap (cont)

Simple solution
- Create a summary “variable” (node) for each allocation statement
- Domain: $2^{(\text{Var} \cup \text{Stmt})} \times (\text{Var} \cup \text{Stmt})$ rather than $2^{\text{Var} \times \text{Var}}$
- Monotonic flow function
  \begin{align*}
  s: & \quad p = \text{new } T; \\
  \text{out}[s] = & \quad \{(p\rightarrow\text{stmt}_s)\} \cup (\text{in}[s] - \{(p\rightarrow x) \forall x\})
  \end{align*}
- Less precise (but finite)

Alternatives
- Summary node for entire heap
- Summary node for each type
- K-limited summary
  - Maintain distinct nodes up to k links removed from root variables

Other issues: Function Calls

Question
- How do function calls affect our points-to sets?
  *e.g.,* \begin{align*}
  p_1 &= & \& x; \\
  p_2 &= & \& p_1; \\
  \ldots \\
  \text{foo}();
  \end{align*}

Be conservative
- Assume that any reachable pointer may be changed
- Pointers can be “reached” via globals and parameters
  - May pass through objects in the heap
- Can be changed to anything reachable or something else
- Can we prune aliases using types?

Problem
- Lose a lot of information
Emami 1994

Overview
- Compute L and R locations to implement flow-sensitive data-flow analysis
- Uses invocation graph for context-sensitivity
- Can be exponential in program size
- Handles function pointers

Characterization of Steensgaard
- Whole program
- Flow-sensitive
- Context-sensitive
- May and must analysis
- Alias representation: points-to
- Heap modeling: one heap variable
- Aggregate modeling: fields and first array element

Using Alias Information

Example: reaching definitions
- Compute at each point in the program a set of \((s,v)\) pairs, indicating that statement \(s\) may define variable \(v\)

Flow functions
- \(s: \ *p = x;\)
  \[\text{out}_{\text{reach}}[s] = \{(s,z) \mid (p \rightarrow z) \in \text{in}_{\text{may-p}}[s]\} \cup (\text{in}_{\text{reach}}[s] - \{(t,y) \forall t \mid (p \rightarrow y) \in \text{in}_{\text{must-p}}[s]\})\]
- \(s: \ x = \ *p;\)
  \[\text{out}_{\text{reach}}[s] = \{(s,x)\} \cup (\text{in}_{\text{reach}}[s] - \{(t,x) \forall t\})\]
- ...
Concepts

Properties of alias analyses

Alias/Pointer Analysis algorithms
- Address Taken
- Steensgaard or FIAlias
- Andersen
- Emami

Flow-insensitive alias algorithms can be specified with constraint equations

Flow-sensitive alias algorithms can be specified with data-flow equations

Function calls degrade alias information
- Context-sensitive interprocedural analysis

Next Time

Assignments
- HW2 due

Lecture
- Midterm review