Definitions

Let r_1, r_2 be references in a program.
r_1 or r_2 may be of the form "x" or "`*p`" "`**p`", "`(*p)->q->i`", etc.

Alias: r_1 are r_2 are aliased if the memory locations accessed by r_1 and r_2 overlap.

Alias Relation: A set of ordered pairs $\{(r_i, r_j)\}$ denoting aliases that may hold at a particular point in a program.
Sometimes called a *may-alias* relation.

Points-to Pair: An ordered pair (r_1, r_2) denoting that one of the memory locations of r_1 may hold the address of one of the memory locations of r_2.
Also written: $r_1 \rightarrow r_2$. Say "r_1 points to r_2".

Points-to Set: $\{(r_i, r_j)\}$: A set of points-to pairs that may hold at a particular point in a program.

Points-To Graph: A directed graph where each *Node* represents one or more memory objects; an *Edge*: $p \longrightarrow q$ means some object in node p may hold a pointer to some object in node q.

Why Is This Difficult?

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

Why Is This Difficult? (cont'd)

5. Interprocedural analysis is crucial. Challenges:
   - callee may modify a pointer used in the caller
   - indirect function calls (i.e., through a function pointer)
   - context-sensitive side-effects of functions
   - formals may be aliased to each other or to globals
   - recursion

6. Compile-time cost
   - Number of variables, $| V |$, can be large
   - Number of alias pairs at a point can be O($| V |^2$)

Common Simplifying Assumptions

1. Aggregate objects: arrays (and perhaps structures) containing pointers
   - *Simple solution*: treat as a single big object!
     Commonplace for arrays.
     Not a good choice for structures? *see Intel paper*
   - Pointer arithmetic is only legal for traversing an array:
     $q = p \pm i$ and $q = \&p[i]$ are handled the same as $q = p$
2. Recursive data structures (lists, trees, graphs, etc.)
   - Compute aliases, not "shape"
     $\Longrightarrow$ Don't prove something is a linked-list or a binary tree
   - k-limiting: only track k or fewer levels of dereferencing
   - Use simplified naming schemes for heap objects (later slide)
3. Control-flow: analyzing different data paths
   - Could ignore the issue and compute a single common solution!
   - *Note*: No consensus on this issue

Naming Schemes for Heap Objects

The Naming Problem: Example 1

```
brillig() {
    // Two distinct objects
    T* p = slithy(n);
    T* q = slithy(m);
}
```

```
T* slithy(int num) {
    // Many objects allocated here
    return new T(...);
}
```

Q. Should we try to distinguish the objects created in `brillig()`?

The Naming Problem: Example 2

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

Q. Can we distinguish the objects created in `makelist()`?

Naming Schemes for Heap Objects

Possible Naming Strategies

1. H : One name for the entire heap
2. H_t : One name per type t (for type-safe languages)
3. H_l : One name per heap allocation site (line number) l
4. H_c : One name per (acyclic) call path c aka "*cloning*"
5. H_f : One name per immediate caller f (or call-site) i.e., *only one-level cloning*

Terminology: Realizable vs. Unrealizable Paths

Definition: Realizable Path

A program path is realizable *iff* every procedure call on the path returns control to the point where it was called (or to a legal exception handler or program exit)

Whole-program Control Flow Graph?

Conceptually extend CFG to span whole program:

- split a call node in CFG into two nodes: CALL and RETURN
- add edge from CALL to ENTRY node of each callee
- add edge from EXIT node of each callee to RETURN

Problem: This produces many unrealizable paths

Focusing only on realizable paths requires a *context-sensitive* analysis.

Flow-sensitive vs. Flow-insensitive Analysis

General definitions

A *flow-sensitive analysis* is one that computes a distinct result for each program point. A *flow-insensitive analysis* generally computes a single result for an entire procedure or an entire program.

Note: A flow-insensitive algorithm effectively *ignores* the order of statements completely!

Alias Analysis

Flow-sensitive : At program point n, compute alias pairs $< a, b >$ that may hold at n for some path from program entry to n.

Flow-insensitive : Compute all alias pairs $< a, b >$ such that a may be aliased to b at *some* point in a program (or function).

Important special cases

- *Local scalar variables*: SSA form gives flow-sensitivity
- *Malloc or new*: Allocates "fresh" memory, i.e., no aliases

Context-sensitive vs. Context-insensitive Analysis

General definitions

A context-sensitive interprocedural analysis computes results that may hold only for realizable paths through a program. Otherwise, the analysis is context-insensitive.

Pointer Analysis

Apply above definitions directly using points-to pairs $< a, b >$.
But important variations exist:

- 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?

Field-sensitive vs. Field-insensitive Analysis

General definitions

A *field-sensitive analysis* is one that tracks distinct behavior for individual fields of a record type. Otherwise, it is *field-insensitive*

Challenges

- *Complexity*: For certain analysis techniques, converts linear representation to worse (perhaps even exponential)
- *Non-type-safe programs*: May have to track behavior at every byte offset within a structure (not just each field)

Some simple, flow-insensitive algorithms

3 popular algorithms

- *Any address*
- *Anderson*, 1994
- *Steensgard*, 1996

Any address

- single points-to set:
  all variables whose address is taken, passed by reference, etc.
- any pointer may point to any variable in this set
- simple, fast, linear-time algorithm
- common choice for function pointers, and for global variables
  - e.g., for initial call graph

Some simple, flow-insensitive algorithms

Example:

```
T *p, *q, *r;
void main() {
  p = new T;
  f();
  g(&p);
  p = new T;
  ... = *p;
}
```

```
void f() {
  q = new T;
  g(&q);
  r = new T;
}
```

```
void g(T** fp) {
  T local;
  if (...)
    fp = &local;
  ...
}
```

Model argument passing and returns with assignment:

- g(&p): Insert:
  - fp = &p
  - p = *fp

Where?

Anderson's Algorithm

Properties:

- Generally the most precise flow- and context-insensitive algorithm
- Compute a single points-to graph for entire program
- Refinement by Burke []: Separate points-to graph for each function
- Cost is $O(n^3)$ for program with n assignments
- Acceptable precision in practice *for optimization*, perhaps not for static analysis tools for security, reliability

Anderson's Algorithm: Conceptual

- *Initialize*: Points-to graph with a separate node per variable
- *Iterate until convergence:*

  At each statement, evaluate the appropriate rule:

p = &x		Add $p \rightarrow x$
p = q	$\forall x$: if $q \rightarrow x$,	add $p \rightarrow x$
*p = q	$\forall x, r$: if $q \rightarrow x$ and $p \rightarrow r$,	add $r \rightarrow x$
p = *q	$\forall x, r$: if $q \rightarrow x$ and $x \rightarrow r$,	add $p \rightarrow r$

- What happens if we process p = q after q = &G?

Anderson's Algorithm: Actual

Actual Algorithm (Sketch):

- Build initial "inclusion constraint graph" and initial points-to sets
- Iterate until converged:
  - Update constraint graph for new points-to pairs *transitive closure*
  - Update the points-to sets according to new constraints

Inclusion Constraint Graph:

Add constraint ("set inclusion") edges for pointer assignments:

Points-to pair	`p = &x`		add loc(x) $\rightarrow p$
Direct constraint	`p = q`		add $q \rightarrow p$
Indirect constraint	`*p = q`	$\forall v$: $v \in pts(p)$,	add $q \rightarrow v$
Indirect constraint	`p = *q`	$\forall v$: $v \in pts(q)$,	add $v \rightarrow p$

Anderson's Algorithm: Cycle Elimination

Cycle in constraint graph:

$$p \rightarrow q \rightarrow r \rightarrow p$$

$$\Rightarrow pts(p) \supseteq pts(q) \supseteq pts(r) \supseteq pts(p)$$

$$\Rightarrow pts(p) = pts(q) = pts(r) = pts(p)$$

$\Rightarrow$ No need to propagate points-to pairs around such cycles!

Offline cycle elimination

- "Off-line Variable Substitution for Scaling Points-To Analysis," Atanas Rountev and Satish Chandra, PLDI 2000.
- Find cycles due to direct pointer copies (direct constraints)
- Collapse each cycle into a single node
- Significantly reduces size of constraint graph Table 2, [PLDI07]
- But many more cycles can be induced by indirect constraint edges:
  $\Rightarrow$ Need cycle elimination during transitive closure ("*online*")

Anderson's Algorithm: Online Cycle Elimination

Several Approaches

- Fähndrich, Foster, Aiken and Su (PLDI '98): Show that cycle elimination is essential for scalability. Limited DFS on each edge insertion to find some cycles.
- Heintze and Tardieu (PLDI 2001): Delay adding indirect constraint edges until querying for aliases. "A million lines of code per second."
- Hardekopf and Lin (PLDI 2007):
  - Lazy cycle detection: Try to detect cycles only when *their effect is observable*: two variables have identical points-to sets
  - Hybrid cycle detection: Build special offline constraint graph with "*ref nodes*" for dereferenced variables (e.g., *p). Use this graph to find cycles. These cycles will expose many (but not all) online cycles *without online graph traversal*. When combined with another cycle detection algorithm, greatly reduces cycle detection costs. [Table 4, Figure 8]

Steensgard's Algorithm

Principles of Prog. Languages, '96

Unification

- Conceptually: restrict every node to only one outgoing edge (on the fly)
- If $p \to x$ and $p \to y$, merge x and y "unify"

$\Rightarrow$ All objects "pointed to" by p are a single equivalence class

Algorithm

- For each statement, merge points-to sets:
  - e.g., p = q:
    Merge two equivalence classes (targets of p and of q)
    This may cause other nodes to collapse!
- Use Tarjan's "union-find" data structure to record equivalence classes:
  Non-iterative algorithm, almost-linear running time: $O(n\alpha(n, n))$
- Again, single solution for entire program

Steensgard vs. Anderson

Consider assignment p = q,
i.e., *only* p *is modified, not* q

Subset-based Algorithms

- Anderson's algorithm is an example
- Add a constraint: Targets of q must be subset of targets of p
- Graph of such constraints is also called "inclusion constraint graphs"
- Enforces *unidirectional flow* from q to p

Unification-based Algorithms

- Steensgard is an example
- Merge equivalence classes: Targets of p and q must be identical
- Assumes *bidirectional flow* from q to p and vice-versa

Which Pointer Analysis Should I Use?

Hind & Pioli, ISSTA, Aug. 2000

Goal

Compared 5 algorithms (4 flow-insensitive, 1 flow-sensitive):

- Any address
- Steensgard
- Anderson
- Burke (like Anderson, but separate solution per procedure)
- Choi et al. (flow-sensitive)

Which Pointer Analysis Should I Use? (cont'd)

Metrics

1. *Precision*: number of alias pairs
2. *Precision of important optimizations*: MOD/REF, REACH, LIVE, flow dependences, constant prop.
3. *Efficiency:* analysis time/memory, optimization time/memory

Benchmarks

23 C programs, including some from SPEC benchmarks

Summary of Results

Hind & Pioli, ISSTA, Aug. 2000

1. *Precision:* *Table 2*

- Steensgard much better than Any-Address (6x on average)
- Anderson/Burke significantly better than Steensgard (about 2x)
- Choi negligibly better than Anderson/Burke

2. *MOD/REF precision:* *Table 2*

- Steensgard much better than Any-Address (2.5x on average)
- Anderson/Burke significantly better than Steensgard (15%)
- Choi very slightly better than Anderson/Burke (1%)

Summary of Results (cont'd)

3. *Analysis cost:* *Table 5*

- Any-Address, Steensgard extremely fast
- Anderson/Burke about 30x slower
- Choi about 2.5x slower than Anderson/Burke

4. *Total cost (analysis + optimizations):* *Table 5*

- Steensgard, Burke are 15% faster than Any-Address!
- Anderson is as fast as Any-Address!
- Choi only about 9% slower