CS 526

Advanced
Compiler
Construction

http://misailo.cs.Illinois.edu/courses/cs526

POINTER ANALYSIS

The slides adapted from Vikram Adve

POINTER ANALYSIS

The slides adapted from Vikram Adve

Pointer Analysis

Pointer and Alias Analysis are fundamental to reasoning about heap manipulating programs (pretty much all programs today).

- **Pointer Analysis:**
  - What objects does each pointer points to?
  - Also called points-to analysis
- **Alias Analysis:**
  - Can two pointers point to the same location?
  - Client of pointer analysis

Example

```
X = 1
P = &X
*P = 2
return X

// What is the value of X?
```

Aliases

Consider references r1 or r2,

- may be of the form "x" or "*p" "**p", "(*p)->q->i"…

Alias: r1 are r2 are aliased if the memory locations accessed by r1 and r2 overlap.

Alias Relation: A set of ordered pairs {(ri, rj)} denoting aliases that may hold at a particular point in a program.

- Sometimes called a *may-alias* relation.

May or Must: A kind of aliasing if it happens optionally or always

- May: e.g., depending on the control flow: `if (b) { p = &q; }`
- Must: determined that they always represents aliases

Points-To Facts

Points-to Pair: An ordered pair (r1, r2) denoting that one of the memory locations of r1 may hold the address of one of the memory locations of r2.

- Also written: r1 → r2, means "r1 points to r2".

Points-to Set: {(ri, rj)} : 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 → q means some object in node p may hold a pointer to some object in node q.

Challenges of Points-To Analysis

- **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
- **Aggregate objects**: structures and arrays containing pointers
- **Recursive data structures** (lists, trees, graphs, etc.) closely related problem: anonymous heap locations
- **Control-flow:** analyzing different data paths
- **Interprocedural:** a location is often accessed from multiple functions; a common pattern (e.g., pass by reference)
- 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

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 (Ghiya, Lavery & Sehr, PLDI 2001)*
- Pointer arithmetic is only legal for traversing an array:

$q = p \pm i$ and $q = \&p[i]$ are handled the same as $q = p$

Common Simplifying Assumptions

Recursive data structures (lists, trees, graphs, …)

Solution: Compute aliases, not "shape"

- Don't prove something is a linked-list or a binary tree (leave that for *shape analysis*)
- k-limiting: only track k or fewer levels of dereferencing
- Use simplified naming schemes for heap objects (later slide)

Common Simplifying Assumptions

Control-flow: analyzing different data paths blows up the analysis time/space

Solution(?): Could ignore the issue and compute a single common result for any path!

No consensus on this issue! (Will discuss later)

Naming Schemes for Heap Objects

The Naming Problem: Example 1

```
int main() {
   // Two distinct objects
   T* p = create(n);
   T* q = create(m);
}

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

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

Naming Schemes for Heap Objects

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()?

Possible Naming Abstractions

H_0 : One name for the entire heap

H_T : One name per type T (for type-safe languages)

H_L : One name per heap allocation site L (line number)

H_C : One name per (acyclic) call path C ("*cloning*")

H_F : One name per immediate caller F or call-site ("*one-level cloning*")

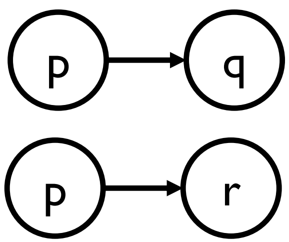
Flow-sensitivity of Analysis

Def. 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.

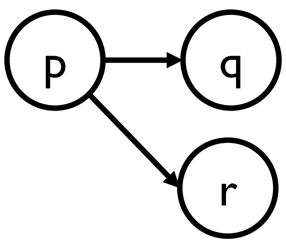
A flow-insensitive algorithm effectively ignores the order of statements!

```
int f(T q, T r){
  T* p;
  …
  p = &q;
  …
  p = &r;
}
```

Flow Sensitive



Flow Insensitive



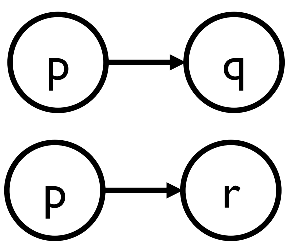
Flow-sensitivity of Analysis

Def. 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.

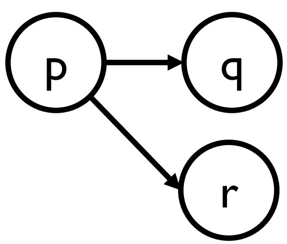
A flow-insensitive algorithm effectively <u>ignores</u> the order of statements!

```
int f(T q, T r){
  T* p;
  if (...)
     p = &q;
  else
     p = &r;
}
```

Flow Sensitive



Flow Insensitive



Flow-Sensitivity of Analysis

Pointer 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
- Read-only fields: e.g., array length

Realizable 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 context-sensitive analysis

Context-Sensitivity of Analysis

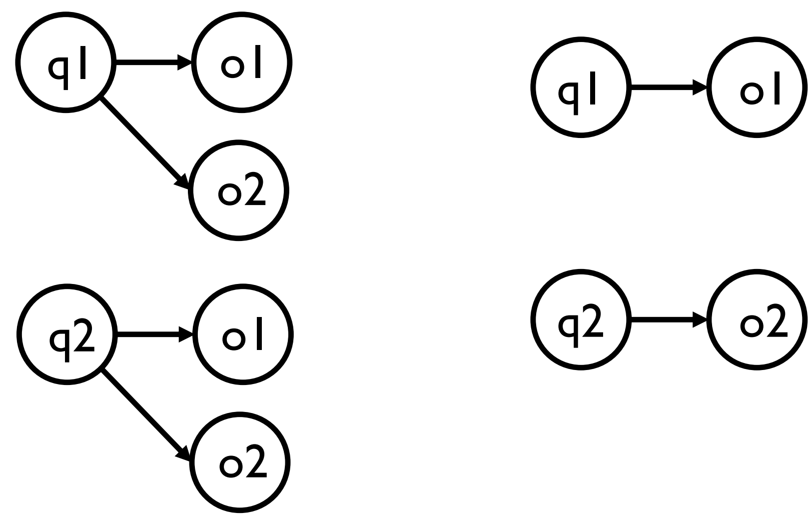
Def. A context-sensitive interprocedural analysis computes results that may hold only for realizable paths through a program. Otherwise, the analysis is context-insensitive.

```
T* identity(T* p) {
  return p;
}

void f1() {
  T* p1 = new T; // Object o1
  T* q1 = identity(p1);
}

void f2() {
  T* p2 = new T; // Object o2
  T* q2 = identity(p2);
}
```

Context Insensitive **Context Sensitive**



Context-Sensitivity of Analysis

Pointer Analysis

Apply the 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?

Object Sensitivity: Context represents each allocation site. Typically offers quite precise context analysis

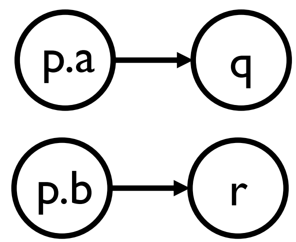
[Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java; Milanova et al. ISSTA 2002]

Field-Sensitivity of Analysis

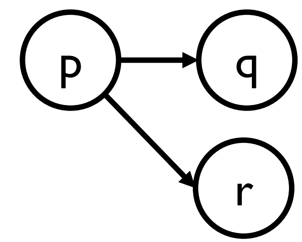
Def. A field-sensitive analysis is one that tracks distinct behavior for individual fields of a record type. Otherwise, it is field-insensitive

```
int f(T q, T r) {
  p.a = &q;
  p.b = &r;
}
```

Field Sensitive



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)

Flow Insensitive Algorithms

3 popular algorithms

- Any address
- Andersen, 1994
- Steensgard, 1996

Acceptable precision in practice for compiler optimization, however perhaps insufficient for static analysis approaches for security, reliability, or bug finding

Any Address Analysis

- 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

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 = new T;
   if (. . .)
      fp = &local;
   . . .
}
```

Model argument passing and returns with assignment:

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

Andersen's Algorithm

- 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

Andersen's Algorithm: Conceptual

Initialize: Points-to graph with a separate node per variable

Iterate until convergence:

At each statement, evaluate the appropriate rule:

Form	**Action**
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 $p \rightarrow r$

Andersen's Algorithm: Actual

1. Build initial "inclusion constraint graph" and initial points-to sets
2. Iterate until converged:
   - Update constraint graph for new points-to pairs
   - Update the points-to sets according to new constraints

Inclusion Constraint Graph: Add constraint for pointer assignments:

Name	**Form**	**Constraint**
Points-to pair	p = &x	$p \supseteq \{x\}$
Direct constraint	p = q	$p \supseteq q$
Indirect constraint	*p = q	$*p \supseteq q$
Indirect constraint	p = *q	$p \supseteq *q$

Andersen's Algorithm: Cycles

Cycle in constraint graph:

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

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

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

Offline cycle elimination:

- Find cycles due to direct pointer copies (direct constraints)
- Collapse each cycle into a single node, significantly reduces size of constraint graph
- But many more cycles can be induced by indirect constraint edges: we need cycle elimination during transitive closure ("*online*")

"Off-line Variable Substitution for Scaling Points-To Analysis,"
Atanas Rountev and Satish Chandra, PLDI 2000.

Online cycle elimination:

- Fähndrich, Foster, Aiken and Su (PLDI '98): Cycle elimination is essential for scalability.
- Heintze and Tardieu (PLDI 2001): "A million lines of code per second."
- Hardekopf and Lin (PLDI 2007)

Steensgard's Algorithm

Unification:

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

Algorithm

1. 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!
2. Use Tarjan's "union-find" data structure to record equivalence classes

Non-iterative algorithm, almost-linear running time: $O(n\alpha(n, n))$

Like Anderson, 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

Alias Analysis

- Alias analysis is a common client of pointer (points-to) analysis
  - **Pointer Analysis:** What objects does each pointer points to?
  - **Alias Analysis:** Can two pointers point to the same location?
- Once we have performed the pointer analysis, it is trivial to compute alias analysis (but not vice versa)

- **Two pointers p and q may alias if pointer-analysis(p) ∩ pointer-analysis(q) ≠ ∅**

Which Pointer Analysis To Use?

Hind & Pioli, ISSTA, Aug. 2000

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)

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

Which Pointer Analysis To Use?

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%)
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 of analysis + optimizations:** (Table 5)
   - Steensgard, Burke are 15% faster than Any-Address!
   - Anderson is as fast as Any-Address!
   - Choi only about 9% slower

Analysis Scalability

	Equality-based	Subset-based	Flow-sensitive
Context-insensitive	• Weihl [32] 1980: < 1 KLOC first paper on pointer analysis • Steensgaard [31] 1996: **1+ MLOC** first scalable pointer analysis	• Andersen [1] 1994: 5 KLOC • Fähndrich et al. [7] 1998: 60 KLOC • Heintze and Tardieu [11] 2001: **1 MLOC** • Berndl et al. [2] 2003: **500 KLOC** first to use BDDs	• Choi et al. [5] 1993: 30 KLOC
Context-sensitive	• Fähndrich et al. [8] 2000: **200K**	• Whaley and Lam [35] 2004: **600 KLOC** cloning-based BDDs	• Landi and Ryder [19] 1992: 3 KLOC • Wilson and Lam [37] 1995: 30 KLOC • Whaley and Rinard [36] 1999: 80 KLOC

Derek Rayside, Points-To Analysis (Summary), 2005

https://www.cs.utexas.edu/~pingali/CS395T/2012sp/lectures/points-to.pdf

Advanced Techniques

- **Shape Analysis**: discovers and reasons about dynamically allocated data structures (e.g., lists, trees, heaps)

- **Escape Analysis:** computes which program locations can access a pointer (across function boundaries)

- **Datalog:** Declarative, constraint-based approach to specify analysis, offers pretty good scalability

  Pointer Analysis; Yannis Smaragdakis; George Balatsouras, Now Publishing, 2015

- Abstract Interpretation Formulation