# Applied Static Analysis

## Pointer Analysis

(The slides are based on slides created by Stephen Chong, Harvard University)

---

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

Summer Term 2019

# Pointer analysis

(Primarily in the context of C(C++))

- Andersen analysis
- Steensgard analysis
- One-level flow
- Pointer analysis for Java

# Pointer analysis

(Recap)

- What memory locations can a pointer expression refer to?
- Alias analysis: When do two pointer expressions refer to the same storage location?
- E.g.,

```
int x;
p = &x;
q = p;
```

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

# Pointer analysis - Aliases

(Recap)

Aliasing can arise due to:

- Pointers

```
int *p, i; p = &i;
```

- Call-by-reference

```
void m(Object a, Object b) { ... } m(x,x); // a and b alias in body of m
m(x,y); // y and b alias in body of m
```

- Array indexing

```
int i,j,a[100];
i = j; // a[i] and a[j] alias
```

# Dimensions of pointer analysis

(Focus of today's lecture.)

- Intraprocedural / **interprocedural**
- Flow-sensitive / **flow-insensitive**
- Context-sensitive / **context-insensitive**
- Definiteness: **may** versus must
- Heap modeling
- Representation

# Flow-sensitive vs flow-insensitive

- **Flow-sensitive pointer analysis** computes for each program point what memory locations pointer expressions may refer to.
- **Flow-insensitive pointer analysis** computes what memory locations pointer expressions may refer to, *at any time in program execution.*

# Possible representations

- Points-to pairs: first element points to the second, e.g.,
  `(p → b), (q → b)`
  `*p` and `b` alias, as do `*q` and `b`, as do `*p` and `*q`
- Equivalence sets: sets that are aliases; e.g.,
  `{*p, *q, b}`

**Context-sensitivity** consists of **qualifying local variables**, and possibly heap abstractions, with context information: the analysis collapses information (e.g., "*What objects this method argument can point to.*") over all possible executions that result in the same context, while separating all information for different contexts.

*Pick your Contexts Well: Understanding Object-Sensitivity; Y. Smaragdakis, M. Bravenboer and O. Lhoták; POPL'11, ACM*

# Context sensitivity

- Goal: improved precision and *scalability*
- Challenge: often bi-modal: either the analysis is precise enough that it only manipulates manageable sets (and then scales very well) or becomes order of magnitude more expensive at the first “imprecision”.
  Additionally, the behavior is hard to predict upfront.
- Forms:
  - call-site-sensitivity
  - object-sensitivity (“standard sensitivity” in case of Java)
  - type-sensitivity
- Generally difficult, but some success: when using context-insensitive analyses as a first step.

# Object- vs. Call-site-sensitivity

- a **1-call-site-sensitive analysis** will distinguish the two call sites of method `foo` in `bar`.
- a **1-object-sensitive analysis** will use the allocation sites of the receiver objects a1 and a2.
- In principle it is not possible to compare the precision of both types of analyses.

```
class A {
  void foo(Object o){…}
}

class Client {
  void bar(A a1, A a2) {
    a1.foo(someObj1);
    …
    a2.foo(someObj2);
  }
}
```

# Modeling memory locations

- We want to describe what memory locations a pointer expression may refer to
- How do we model memory locations?
  - For global variables, no trouble, use a single “node”
  - For local variables, use a single “node” per context; i.e., just one node if context insensitive.
  - For dynamically allocated memory we have to handle a potentially unbounded set of locations which are created at runtime; we need to model locations with some *finite abstraction*

# Modeling dynamic memory locations

- For each allocation statement, use one node per context (most common solution)
- One node for each type
- Nodes based on analysis of “shape” of heap
- ...

# Let's consider flow-insensitive may pointer analysis

- Assume program consists of statements of form:
  `p = &a` (address of, includes allocation statements)
  `p = q`
  `*p = q`
  `p = *q`

- Assume pointers `p,q ∈ P` and address-taken variables `a,b ∈ A` are disjoint

- We want to compute relation P∪A → $\mathscr{P}$(A)
  (Essentially points to pairs.)

# Andersen-style pointer analysis

- View pointer assignments as subset constraints
- Use constraints to propagate points-to information

| 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) |

# Andersen-style pointer analysis

(Example I)

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

```
p = &a;    p ⊇ {a}   pts(p) = {a,b}
q = p;     q ⊇ p     pts(q) = {a,b}
p = &b;    p ⊇ {b}   pts(r) = {a,b}
r = p;     r ⊇ p     pts(a) = ∅
                     pts(b) = ∅
```

# Andersen-style pointer analysis

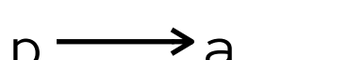
(Example II)

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

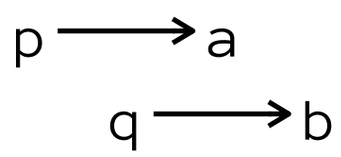
| | | |
|---|---|---|
| `p = &a;` | `p ⊇ {a}` | `pts(p) = {a}` |
| `q = &b;` | `q ⊇ {b}` | `pts(q) = {b}` |
| `*p = q;` | `*p ⊇ q` | `pts(r) = {c}` |
| `r = &c;` | `r ⊇ {c}` | `pts(s) = {a}` |
| `s = p;` | `s ⊇ p` | `pts(t) = {b,c}` |
| `t = *p;` | `t ⊇ *p` | `pts(a) = {b,c}` |
| `*s = r;` | `*s ⊇ r` | `pts(b) = ∅` |
| | | `pts(c) = ∅` |

# Precision?

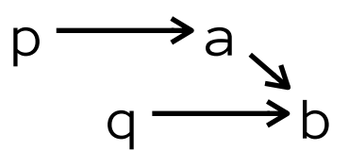
```
p = &a;
```



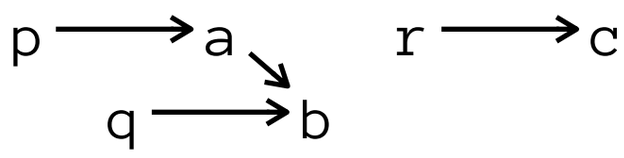

```
q = &b;
```



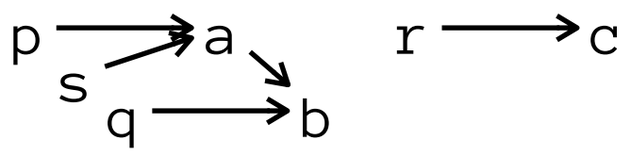

```
*p = q;
```



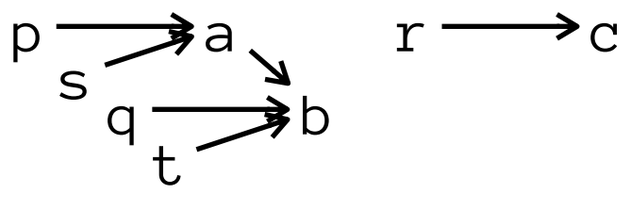

```
r = &c;
```



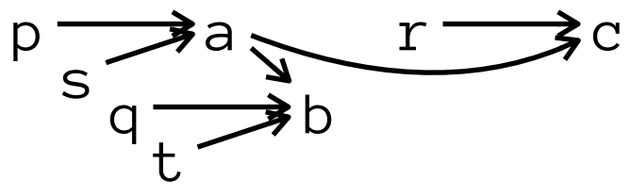

```
s = p;
```




```
t = *p;
```




```
*s = r;
```




```
pts(p) = {a}

pts(q) = {b}

pts(r) = {c}

pts(s) = {a}

pts(t) = {b,c}

pts(a) = {b,c}

pts(b) = ∅

pts(c) = ∅
```

# Andersen-style as graph closure

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

| Constraint type | Assignment | Constraint | Meaning | Edge |
|---|---|---|---|---|
| Base | a = &b | a ⊇ {b} | loc(b) ∈ pts(a) | no edge |
| Simple | a = b | a ⊇ b | pts(a) ⊇ pts(b) | b → a |
| Complex | a = *b | a ⊇ b* | ∀v∈pts(b).pts(a) ⊇ pts(v) | no edge |
| Complex | *a = b | *a ⊇ b | ∀v∈pts(a).pts(v) ⊇ pts(b) | no edge |

- Each node has an associated points-to set
- Compute transitive closure of graph, and add edges according to complex constraints

# Worklist algorithm

Initialize graph and points to sets using base and simple constraints.

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

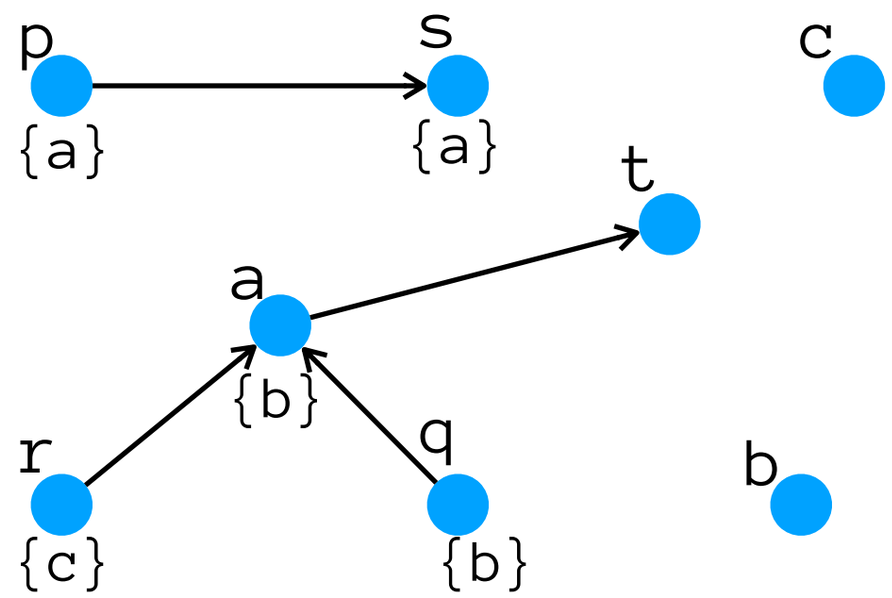
    for each edge v→q do

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

# Andersen-style pointer analysis

## (Example II as graph - Step 1)

```
p = &a;    p ⊇ {a}
q = &b;    q ⊇ {b}
*p = q;    *p ⊇ q
r = &c;    r ⊇ {c}
s = p;     s ⊇ p
t = *p;    t ⊇ *p
*s = r;    *s ⊇ r
```

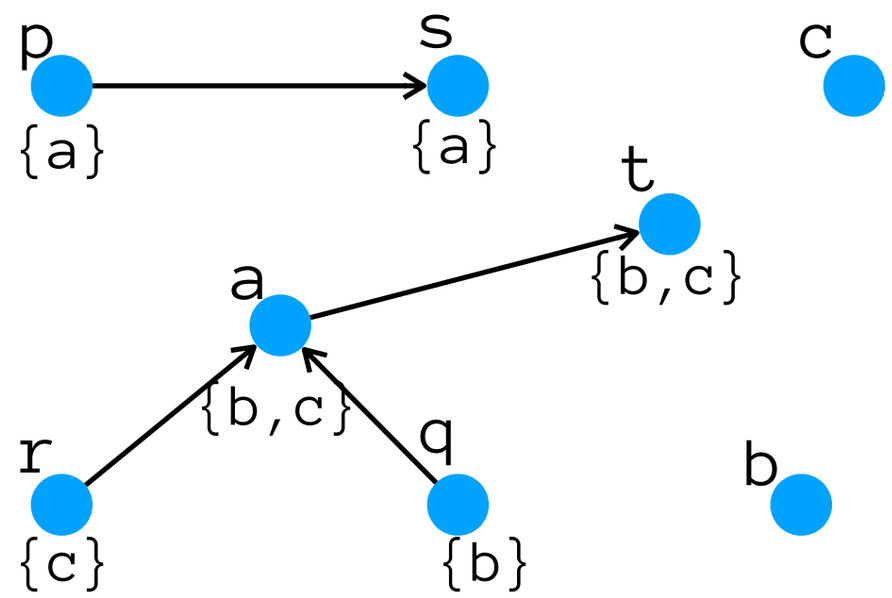



Initial worklist `W`: `{p q r s a}`

# Andersen-style pointer analysis

## (Example II as graph - Step 2)

```
p = &a;    p ⊇ {a}
q = &b;    q ⊇ {b}
*p = q;    *p ⊇ q
r = &c;    r ⊇ {c}
s = p;     s ⊇ p
t = *p;    t ⊇ *p
*s = r;    *s ⊇ r
```



# Performance optimization for Andersen-style analysis: Cycle elimination

Andersen-style pointer analysis is O(n3), for number of nodes in graph (Actually, quadratic in practice…)

- Detect strongly connected components (SCC) in points-to graph, collapse to single node.
  All nodes in an SCC will have same points-to relation at end of analysis
- How to detect cycles efficiently?
  Some reduction can be done statically, some on-the-fly as new edges added.
  (For details: The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, Hardekopf and Lin, PLDI 2007)

# Steensgaard-style analysis

- Also a constraint-based analysis, but **uses equality constraints** instead of subset constraints.

- Originally phrased as a type-inference problem
  Less precise than Andersen-style, (at least) for C more scalable.

| 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) |

- Can be efficiently implemented using Union-Find algorithm.

- Nearly linear time; each statement needs to be processed just once.

# One-level flow

- Observation: common use of pointers in C programs is to pass the address of composite objects or updatable arguments; multi-level use of pointers not as common
- Uses unification (like Steensgaard) but avoids unification of top-level pointers (pointers that are not themselves pointed to by other pointers).
  I.e., use Andersen's rules at top level, Steensgaard's elsewhere.
- Precision close to Andersen's, scalability close to Steensgaard's – If the observation holds; which is not the case for Java, C++, …

# Pointer analysis in Java

- Different languages use pointers differently: *Scaling Java Points-To Anlaysis Using SPARK*, **Lhotak & Hendren,** CC'03
- Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation & creates stack-directed pointers; malloc creates heap-directed pointers
- Java allows no stack-directed pointers, many more dynamic allocation sites than similar-sized C programs
- Java is strongly typed which limits set of objects a pointer can point to; this helps precision
- Dereference in Java only through field store and load
- …

# Object-sensitive pointer analysis

- Parameterized object sensitivity for points-to analysis for Java; **Milanova, Rountev, and Ryder**; ACM Trans. Softw. Eng. Methodol., 2005.
  - Context-sensitive interprocedural pointer analysis
  - For context, use stack of receiver objects
- Context-sensitive points-to analysis: is it worth it?; **Lhotak and Hendren**, CC'06:
  - Object-sensitive pointer analysis more precise than call-stack contexts for Java
  - Likely to scale better