Pointer Analysis: Inclusion-based & Unification-based

(Target: 15 Minutes)

1 1. Introduction & The "Why" (3 Minutes)

Today I will present **Pointer Analysis**, specifically focusing on the inclusion-based techniques described in Chapter 11.

To start, I want to clarify *why* this is such a fundamental problem in compiler construction. It all comes down to one word: **Safety**. When a compiler tries to optimize code—say, by keeping a variable in a register or removing dead code—it must guarantee that the program's behavior doesn't change. Pointers threaten this guarantee because they introduce **Aliasing**.

Write on Whiteboard:

- **Header:** Motivation: Aliasing
- **Code Example:**

```
x = 10;
*p = 5;      // The "Unknown" Store
y = x;
// Can we optimize this to y = 10?
```

Consider this snippet. We assign 10 to x. Then we store 5 into the address p. Finally, we read x. A naive compiler might look at line 3 and say: "Well, x is 10, so let's just replace $y = x$ with $y = 10$."

But can we do that? If p happens to point to x, then line 2 overwrites x with 5. If we optimized y to 10, we would have broken the program.

- **The Goal:** Compute a set of abstract locations that p might point to.
- **The Optimization:** If x is NOT in that set, we can optimize safely. If x IS in that set, we must be conservative.

2 2. Concepts: Locations & Constraints (3 Minutes)

Before we look at the algorithms, we need to define our **Domain**. What are we analyzing? We model memory as a set of **Abstract Locations**. Obviously, every global and local variable (like x or y) is a location. But we also need to handle dynamic memory (malloc).

Since a program can call ‘malloc‘ infinite times in a loop, we cannot track every single block. Instead, we group all blocks allocated at a specific line of code into one **Abstract Object**.

Write on Whiteboard:

- List the Abstract Locations:
  1. Variables (x, y, z)
  2. Allocation Sites $(S_1, S_2 \ldots)$
- Define the goal: Compute $pt(p)$, the subset of locations p may point to.

3 3. Andersen's Analysis (Inclusion-based) (6 Minutes)

The most common solution is **Andersen's Analysis**. This relies on **Subset Constraints**. The intuition is that assignments create a flow of data. If I say $p = q$, I am saying that p can now see everything q sees.

Write on Whiteboard:

- Draw the Rules Table:

Stmt	**Code**	**Constraint**
Address	$p = \&x$	$\{x\} \subseteq pt(p)$
Copy	$p = q$	$pt(q) \subseteq pt(p)$
Load	$p = *q$	$\forall v \in pt(q) : pt(v) \subseteq pt(p)$
Store	$*p = q$	$\forall v \in pt(p) : pt(q) \subseteq pt(v)$

The first two rules are simple. The complexity lies in the Load and Store rules.

- Take $p = *q$. We are dereferencing q.
- Since q holds addresses, we must look at every location v that q *might* point to.
- For each of those v's, we copy their contents into p.

This is dynamic. As our knowledge of $pt(q)$ grows, the constraints generated by this rule also grow.

The Algorithm: Constraint Graph

We solve this using a **Constraint Graph**. Nodes are variables, and an edge from $q \to p$ means $pt(q) \subseteq pt(p)$.

Whiteboard Action:

- Draw 4 nodes in a diamond shape: p, q, x, y.
- Illustrate the Worklist approach:
  1. Start with base constraints (e.g., $p = \&x \to$ add x to $pt(p)$).
  2. If $pt(p)$ changes, propagate new info to neighbors via outgoing edges.
  3. Repeat until Fixed Point.

Complexity: $O(N^3)$. This is due to the dynamic Load/Store rules. A change in one pointer set can trigger a ripple effect that adds edges to the graph, triggering more propagation.

4 4. Steensgaard's Analysis (Unification) (3 Minutes)

$O(N^3)$ is acceptable for small programs, but too slow for massive codebases (like the Linux Kernel). This brings us to **Steensgaard's Analysis**.

Steensgaard asked: "What if we sacrifice precision for speed?" Instead of allowing data to flow in one direction ($q \subseteq p$), Steensgaard forces the sets to be identical ($q = p$). If we assign $p = q$, we treat them as the **same node** in the graph.

Write on Whiteboard:

- **Steensgaard:** Unification (=) not Subset ($\subseteq$).
- Draw the Rules Table for Comparison:

Stmt	**Code**	**Constraint (Unification)**
Address	$p = \&x$	$pt(p) = \{x\}$ (Merge x into p's points-to set)
Copy	$p = q$	$pt(p) = pt(q)$ (Merge sets p and q)
Load	$p = *q$	$pt(p) = *pt(q)$ (Merge p with whatever q points to)
Store	$*p = q$	$*pt(p) = pt(q)$ (Merge whatever p points to with q)

Key Difference in Logic:

- In Andersen's, if p points to $\{x, y\}$, the constraint $*p = q$ means "Updates to $*p$ flow into x **AND** y."

- In Steensgaard's, we cannot handle "AND." We can only handle one set. So, if p points to x and y, Steensgaard **unifies** x and y. They become the same node in the graph.

Whiteboard Action:

- **Diagram:**
  - **Andersen:** Node p has two outgoing arrows to x and y.
  - **Steensgaard:** Nodes x and y are merged into a single blob. Node p has one arrow to that blob.

This turns the problem from Graph Reachability into **Union-Find**.

- **Complexity:** Almost Linear ($O(N\alpha(N))$).
- **Downside: False Positives**. Because we merged x and y, the analysis might report that a pointer intended for x also points to y. This is a loss of precision.

5 5. Conclusion (<1 Minute)

Summary:

- We defined **Aliasing** as the core problem preventing optimization.
- We explored **Andersen's method** (Inclusion-based), which is precise ($O(N^3)$) and uses a constraint graph.
- We contrasted it with **Steensgaard's method** (Unification-based), which is fast ($O(N\alpha(N))$) but less precise due to merging.

In modern compilers (LLVM/GCC), we typically use inclusion-based analysis heavily optimized with **Cycle Detection** to ensure scalability.

Appendix: Quick Reference

Comparison Table

Analysis	**Constraint**	**Complexity**	**Data Structure**
Andersen	Subset ($\subseteq$)	$O(n^3)$	Constraint Graph
Steensgaard	Equality ($=$)	$O(n\alpha(n))$	Union-Find

Key Terminology

- **Abstract Location:** Represents a variable or a ‘malloc‘ site.
- **Flow-Insensitive:** The order of statements doesn’t matter (sets accumulate).
- **Fixed Point:** When propagating sets produces no new changes.
- **Cycle Detection:** Optimization for Andersen’s (all nodes in a cycle are merged).