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 $(S_1, S_2 \dots)$
- 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** (Section 11.2). This relies on **Subset Constraints** (inclusion). The intuition is that assignments create a flow of data. If I say $p = q$, I am saying that p can now point to everything q points to.

Write on Whiteboard:

- **Domain:** $pt(p) = [\![p]\!]$ is the set of abstract cells p may point to.
- **Rules:**

Stmt	**Code**	**Constraint**
Allocation	$X = \text{alloc } P$	alloc-$i \in [\![X]\!]$
Address	$X_1 = \&X_2$	$X_2 \in [\![X_1]\!]$
Copy	$X_1 = X_2$	$[\![X_2]\!] \subseteq [\![X_1]\!]$
Load	$X_1 = *X_2$	$\forall c \in \text{Cell} : c \in [\![X_2]\!] \Rightarrow [\![c]\!] \subseteq [\![X_1]\!]$
Store	$*X_1 = X_2$	$\forall c \in \text{Cell} : c \in [\![X_1]\!] \Rightarrow [\![X_2]\!] \subseteq [\![c]\!]$

The complexity lies in the Load and Store rules. They rely on the points-to set of the pointer being dereferenced. As our knowledge of $[\![X_2]\!]$ grows, we discover new constraints (dynamic edges in the constraint graph).

- **Complexity:** $O(N^3)$ in the worst case.

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

Alternatively, we have **Steensgaard's Analysis** (Section 11.3). This is coarser but faster. Instead of subsets ($\subseteq$), it uses **Unification** ($=$) and equivalence classes. It treats assignments as bidirectional.

Notation:

- $[\![X]\!]$ is a **term variable** representing the abstract cell that X points to.
- We use a constructor, let's call it $\uparrow$ (or t), to represent "pointer to".

Stmt	**Code**	**Constraint**
Allocation	$X = \text{alloc } P$	$[\![X]\!] =\uparrow [\![\text{alloc-}i]\!]$
Address	$X_1 = \& X_2$	$[\![X_1]\!] =\uparrow [\![X_2]\!]$
Copy	$X_1 = X_2$	$[\![X_1]\!] = [\![X_2]\!]$
Load	$X_1 = *X_2$	$[\![X_2]\!] =\uparrow \alpha \wedge [\![X_1]\!] = \alpha$
Store	$*X_1 = X_2$	$[\![X_1]\!] =\uparrow \alpha \wedge [\![X_2]\!] = \alpha$

Key Difference:

- In Andersen's (Copy): $X_1 = X_2 \implies [\![X_2]\!] \subseteq [\![X_1]\!]$. (Flow is directional).
- In Steensgaard's (Copy): $X_1 = X_2 \implies [\![X_1]\!] = [\![X_2]\!]$. (Flow is bidirectional/unified).

This unification merges the sets of locations pointed to by X_1 and X_2, losing precision but running in almost linear time $O(N\alpha(N))$.

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).