Introduction (2 Minutes)

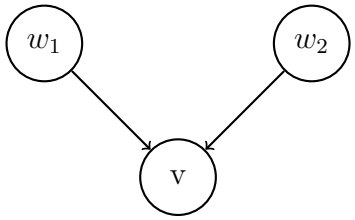
Today I will present the core concepts of **Dataflow Analysis** using **Monotone Frameworks**. We will start by establishing the general mathematical structure that allows us to solve flow-sensitive problems like Constant Propagation or Liveness Analysis.

However, standard frameworks have limitations. Specifically, they require lattices of finite height to guarantee termination, and they often ignore the specific path taken through the control flow graph.

Therefore, in the second half of the presentation, I will introduce **Widening and Narrowing** to handle infinite lattices (like Interval Analysis) and **Path Sensitivity** to handle correlated branches.

Whiteboard Action:

- Write the title: **Dataflow Analysis**.
- Draw a simple CFG node v with predecessors w_1, w_2.



Part 1: Monotone Frameworks (Detailed Breakdown)

To perform dataflow analysis systematically, we use a **Monotone Framework**. A framework is essentially a pair: a Control Flow Graph (CFG) representing our program, and a **Complete Lattice** L representing the abstract information we want to track.

For every node v in the CFG, we assign a **constraint variable**, denoted as $[\![v]\!]$. This variable holds an element of our lattice L—an 'abstract state'—associated with that program point.

The core of the analysis is defining how this information flows and changes. We do this by setting up a **Dataflow Constraint** for every node. The constraint equation generally looks like this:

Whiteboard Action:

- Write: $[\![v]\!] = f_v(JOIN(v))$

This equation has two distinct parts that do very different jobs: the **JOIN** operation and the **Transfer Function** (f_v).

First, let's look at **JOIN**. In a control flow graph, execution paths merge. We might

arrive at node v from multiple different places. Since we are doing static analysis, we must be conservative and consider *all* possible incoming paths. The JOIN operation combines the abstract states from these neighbors into a single, safe state.

The definition of JOIN depends entirely on the **direction** of our analysis:

Whiteboard Action:

- Write:
  - **Forward Analysis:** $JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$ (Combine information from the past).
  - **Backward Analysis:** $JOIN(v) = \bigsqcup_{w \in succ(v)} [\![w]\!]$ (Combine information from the future).

Once we have joined the incoming information, we apply the **Transfer Function**, denoted as f_v (or sometimes t_v).

The Transfer Function models the *semantics* of the statement at node v. It specifies how the execution of that specific statement transforms the abstract state.

For example, if we are doing Sign Analysis and node v is the statement 'x = -5':

- The JOIN operation tells us the state of the world right before we execute the statement.
- The Transfer Function f_v takes that state and updates it to reflect that 'x' is now definitely Negative.

So, in summary: JOIN merges the paths to get us to the node, and the Transfer Function calculates the effect of the node itself.

We also distinguish between **May** and **Must** analysis based on the lattice ordering and the join operator:

1. **May Analysis** (e.g., Liveness): Uses union ($\cup$) as the join. It over-approximates.
2. **Must Analysis** (e.g., Available Expressions): Uses intersection ($\cap$) as the join. It under-approximates.

If the lattice L has finite height and the functions f_v are **monotone**, we can use Kleene's Fixed-Point Theorem. We simply iterate from the bottom element $\bot$ until the values stabilize to the least fixed point, $lfp(f)$.

Whiteboard Action:

- Draw the classification table:

	Forward	Backward
May ($\cup$)	Reaching Defs	Live Vars
Must ($\cap$)	Avail. Expr	Very Busy Expr

Part 2: Widening and Narrowing (5 Minutes)

The framework I just described works perfectly if our lattice has *finite height*. But what if we want to track ranges of integers? We use the **Interval Lattice**:

$$Interval = lift(\{[l, h] \mid l, h \in N \wedge l \leq h\})$$

Here, N includes $-\infty$ and ∞. The problem is that this lattice has **infinite height**.

Whiteboard Action:

- Draw the infinite chain: $[0, 0] \sqsubseteq [0, 1] \sqsubseteq [0, 2] \ldots$

Consider this example from the course book:

```
y = 0; x = 7; x = x + 1;
while(input) {
    x = 7; x = x + 1;
    y = y + 1;
}
```

Without special handling, the analysis of 'y' inside the loop would produce the sequence $[0, 0], [0, 1], [0, 2], \ldots$ and never terminate.

To fix this, we use **Widening** (∇). We introduce a set of constants B (e.g., $\{-\infty, 0, 1, 7, \infty\}$). If a bound is unstable, we widen it immediately to the next value in B. For our example, 'y' jumps from $[0, 1]$ directly to $[0, \infty]$.

This guarantees termination, but it overshoots. In this example, widening might conclude that x is $[7, \infty]$, which is safe, but very imprecise because we can see that x is always 8 inside the loop.

To recover this precision, we use **Narrowing**. This is where we must understand the math.

The result of widening, let's call it f_∇, is a **safe approximation**. Mathematically, this means two things: 1. It is above the least fixed point: $lfp(f) \sqsubseteq f_\nabla$. 2. It is a *post-fixed point*: $f(f_\nabla) \sqsubseteq f_\nabla$.

Because of property #2, if we apply the transfer function f one more time to our result, we get a value $f(f_\nabla)$ that is **smaller** (more precise) than what we started with.

But is it still safe? Yes, because our transfer function f is **monotone**. Since $lfp(f) \sqsubseteq f_\nabla$, applying f to both sides implies $f(lfp(f)) \sqsubseteq f(f_\nabla)$. Since $f(lfp(f))$ is just $lfp(f)$, we prove that our new result is still a valid upper bound.

In our example, applying narrowing brings x from $[7, \infty]$ back down to $[8, 8]$, perfectly restoring the lost information."

Whiteboard Action:

- Write the Narrowing Logic:

$$\underbrace{lfp(f) \sqsubseteq f(f_\nabla)}_{\text{Still Safe}} \sqsubseteq \underbrace{f_\nabla}_{\text{More Precise}}$$

- Show the Example Result:

$$x : [7, \infty] \xrightarrow{\text{Narrowing}} [8, 8]$$

Path Sensitivity (4 Minutes)

Standard monotone frameworks are **path-insensitive**. They merge information from all branches using the Join operator ($\bigsqcup$), losing the correlation between variables.

Consider a case where we open a file only if a `flag` is set, and close it only if the `flag` is set. A standard analysis merges the paths and might conclude we are closing a file that might not be open.

To fix this, we introduce **Path Sensitivity**. We can do this in two ways:

1. Control Sensitivity (Assertions) We can explicitly model conditions using an `assert(E)` statement. If we have `if (x > 0)`, we insert `assert(x > 0)` in the true branch and `assert(!(x > 0))` in the false branch. In Interval Analysis, `assert(x > 0)` refines the interval of x. If x was $[-\infty, \infty]$, it becomes $[1, \infty]$ in that branch.

2. Relational Analysis (Path Contexts) For stronger properties (like the file-flag correlation), we use a **Relational Analysis**. We replace our lattice L with a map lattice:

$$L' = Path \to L$$

Here, $Path$ is a set of path contexts (e.g., predicates like $\{flag = 0, flag \neq 0\}$). Instead of one abstract state per node, we maintain a separate state for every context.

Whiteboard Action:

- Write lattice def: $L' = Path \to \mathcal{P}(\{open, closed\})$
- Example constraint for `open()`:

$$[\![open()]\!] = \lambda p.\{open\}$$

- Example constraint for `flag=0`:

$$[\![flag = 0]\!] = [flag = 0 \mapsto \bigcup_p JOIN(v)(p), \quad flag \neq 0 \mapsto \emptyset]$$

By separating the dataflow based on the ‘flag‘ context, we can prove that in the ‘flag $\neq$ 0‘ context, the file is open, and in the ‘flag $=$ 0‘ context, the file is closed. This solves the precision problem.

Conclusion (< 1 Minute)

In summary, Monotone Frameworks provide the theoretical foundation for static analysis using lattices and fixed-point algorithms.

- When the lattice height is infinite, we apply **Widening** to ensure termination and **Narrowing** to restore precision.
- When control flow correlations matter, we use **Path Sensitivity** (via assertions or relational path contexts) to distinguish feasible execution paths.

Thank you. I am happy to take any questions.