

# Oral Exam Script: Program Analysis (Expanded)

Candidate

January 11, 2026

## Topic 1: Type Analysis (Chapter 3)

**1. Overview** I will discuss Type Analysis for the TIP language. Since TIP lacks explicit type declarations, we must infer types to ensure safety. We want to prevent errors like applying arithmetic to pointers or dereferencing integers.

Our goal is "typability": we generate equality constraints from the AST, and if a solution exists, the program is safe.

**2. The Type Language Grammar** First, we must define the grammar for types, denoted as  $\tau$ . The grammar includes:

- **Basic Types:**  $\text{int}$
- **Pointers:**  $\uparrow \tau$  (a pointer to type  $\tau$ )
- **Functions:**  $(\tau_1, \dots, \tau_n) \rightarrow \tau$
- **Type Variables:**  $\alpha, \beta, \dots$  (used for inference)
- **Recursive Types:**  $\mu \alpha. \tau$
- **Records:**  $\{id_1 : \tau_1, \dots, id_n : \tau_n\}$

**Example: Recursive Types** Standard finite types cannot describe recursive structures. We use **Regular Types** (infinite trees) represented by the  $\mu$  binder.

**Example:** A recursive function `foo` might have the type:

$$\mu t. (\uparrow \text{int}, t) \rightarrow \text{int}$$

Here,  $t$  represents the function type itself appearing inside the arguments.

**Example: Polymorphism** We also see **Parametric Polymorphism** via free type variables.

**Example:** Consider a function `store(a,b) { *b = a; return 0; }`. Its inferred type is:

$$(\alpha, \uparrow \alpha) \rightarrow \text{int}$$

This means it works for *any* type  $\alpha$ , as long as the second argument is a pointer to the first.

**3. Constraint Generation** We assign a type variable  $[[E]]$  to every node  $E$  in the AST and generate constraints. Some key rules include:

- **Literals:**  $[[\text{int}]] = \text{int}$  and  $[[\text{input}]] = \text{int}$ .
- **Allocations:** For an allocation ‘alloc E’:

$$[[\text{alloc } E]] = \uparrow [[E]]$$

- **Pointer Operations:** For a dereference ‘\*E’:

$$[[E]] = \uparrow [[*E]]$$

This enforces that  $E$  is a pointer and links it to the result type.

- **Null:** The null pointer is flexible. We generate a **fresh** type variable  $\alpha$ :

$$[[\text{null}]] = \uparrow \alpha$$

- **Functions:** For a function definition  $X(X_1 \dots X_n)\{\text{return } E\}$ :

$$[[X]] = ([[X_1]], \dots, [[X_n]]) \rightarrow [[E]]$$

**4. Unification Algorithm** To solve these constraints, we use the **Unification Algorithm** based on Union-Find. We iterate through constraints and UNIFY terms.

- **Priority:** Proper types (like `int`) beat type variables. If we unify  $\alpha = \text{int}$ , the representative becomes `int`.
- **Cycles:** If we encounter a cycle like  $X = \uparrow X$ , the solver constructs a recursive type  $(\mu \dots)$  rather than crashing.

**5. Record Types** For records, we don't know which fields exist. We handle this using an "absent field" type, denoted  $\circ$ .

**Field Lookup Rule ( $E.X$ ):** 1. We assume the record *might* contain all fields  $f_1 \dots f_m$ . 2. We generate  $[[E]] = \{f_1 : \gamma_1, \dots, f_m : \gamma_m\}$ . 3. If the field is accessed,  $\gamma = [[E.X]]$ . If not,  $\gamma$  is a fresh variable. 4. **Crucial Check:** After unification, we verify that  $[[E.X]] \neq \circ$  to ensure the field actually exists.

**6. Limitations & Solutions** Finally, we must acknowledge the limitations of this specific analysis:

1. **Flow-Insensitivity** The analysis ignores execution order. It assumes a variable has exactly one type throughout the entire function.

**Example:** If you assign an integer to  $x$  and later assign a pointer to  $x$ , this analysis will reject the program, even if the usages are safe in temporal order. To fix this, we would need flow-sensitive analysis (like SSA).

2. **Polymorphism (The "Monomorphic" Problem)** Our current analysis is **monomorphic**.

**Problem:** Consider the identity function `id(x) { return x; }`. If we call it with an `int`, the type variable for  $x$  unifies with `int`. If we later call it with a `pointer`, unification fails because `int`  $\neq$  `pointer`. The function cannot be reused for different types.

**Solution: Let-Polymorphism** To solve this, we can use **Let-Polymorphism** (common in languages like OCaml or ML).

- **Generalization:** When a function is defined (e.g., in a `let` binding), we generalize its free type variables into a "type scheme" (e.g.,  $\forall \alpha. \alpha \rightarrow \alpha$ ).
- **Instantiation:** Every time the function is *used*, we replace the quantified variables ( $\alpha$ ) with **fresh** type variables.
- **Result:** One use of `id` gets fresh variables that unify with `int`, and the next use gets *different* fresh variables that unify with `pointer`.

**Trade-off:** While flexible, this increases the worst-case complexity of the analysis from almost-linear to **exponential**.

3. **Runtime Errors** Lastly, typability does not guarantee complete safety. It does not catch null pointer dereferences, reading uninitialized variables, or division by zero.

## Topic 2: Dataflow Analysis & Monotone Frameworks (Detailed)

**1. The Monotone Framework** I will now discuss Dataflow Analysis using the **Monotone Framework**. Instead of inventing a new algorithm for every analysis, we define a general framework on a **Complete Lattice**  $(L, \sqsubseteq)$ .

The framework requires two main ingredients:

1. A **Transfer Function**  $f : L \rightarrow L$  for each node, which models how an instruction affects the state. Crucially, this function must be **monotone**: if  $x \sqsubseteq y$ , then  $f(x) \sqsubseteq f(y)$ .
2. A **Join Operator** ( $\sqcup$ ) to merge information from different paths (e.g., at the end of an ‘if’ statement).

We solve the system using the **Work-List Algorithm**. We initialize all nodes to  $\perp$  (bottom), put them in a list, and iteratively update them using the transfer functions. **Guarantee:** If the lattice has **Finite Height** and functions are monotone, this algorithm is guaranteed to terminate with the unique Least Fixed Point.

**2. The Four Classic Analyses** The framework categorizes standard analyses based on **Direction** (Forward/Backward) and **Quantification** (May/Must).

- **Reaching Definitions (Forward, May)**

- **Goal:** Find which assignments (definitions) might reach a specific program point without being overwritten.
- **Lattice:** Sets of definition pairs  $(Var, Label)$ .
- **Join:** Union ( $\cup$ ). Since it is a “May” analysis, we want *any* definition that reaches along *any* path.
- **Use:** Def-use chains for optimization.

- **Available Expressions (Forward, Must)**

- **Goal:** Determine which expressions have definitely been computed and not modified.
- **Join:** Intersection ( $\cap$ ). Since it is a “Must” analysis, an expression is only available if it comes from *all* incoming paths.

- **Use:** Common Subexpression Elimination (CSE).
- **Live Variables (Backward, May)**
  - **Goal:** Determine if a variable holds a value that will be read (used) in the future before being overwritten.
  - **Direction:** Backward. We start from the "use" and propagate back to the "def".
  - **Join:** Union ( $\cup$ ). If a variable is live on *any* future path, it is live here.
  - **Use:** Dead Code Elimination (remove assignments to non-live variables).
- **Very Busy Expressions (Backward, Must)**
  - **Goal:** Find expressions that are evaluated on *all* future paths from the current point.
  - **Join:** Intersection ( $\cap$ ).
  - **Use:** Code Hoisting (moving computations earlier to save size).

**3. Handling Infinite Lattices (Widening)** Standard dataflow analysis assumes the lattice has **finite height** to guarantee termination. However, more powerful analyses, like **Interval Analysis** (which tracks variable ranges  $[min, max]$ ), operate on lattices of **Infinite Height**.

**The Problem:** Consider a loop ‘while(true)  $x++$ ’. The intervals would evolve:  $[0, 0] \rightarrow [0, 1] \rightarrow [0, 2] \dots$ . The chain never stabilizes, so the standard algorithm loops forever.

**The Solution:** We use **Widening ( $\nabla$ )**.

- **Mechanism:**  $\nabla$  is an acceleration operator. If it sees bounds growing unstable (e.g.,  $0 \rightarrow 1 \rightarrow 2$ ), it jumps directly to infinity ( $[0, \infty]$ ).
- **Result:** This guarantees termination but loses precision.
- **Refinement ( $\Delta$ ):** After widening stabilizes, we apply **Narrowing ( $\Delta$ )**. We run a few standard iterations downwards from infinity to recover tighter bounds while remaining sound.

**4. Path Sensitivity** Standard dataflow analysis is typically **path-insensitive**. It merges information from all incoming branches at join points (like the end of an ‘if/else’). While this ensures efficiency, it frequently results in a loss of precision. To fix this, we have three distinct techniques.

1. **Control Sensitivity (Refining with Guards)** Standard analysis propagates the abstract state into both branches of a conditional equally. Control sensitivity exploits the **branch conditions** (guards) to filter these states.

- **Mechanism:** We treat the branch condition as an *assertion*. If we have ‘if  $(x > 0)$ ’, we restrict the abstract value of  $x$  in the “true” branch to exclude non-positive numbers.
- **Benefit:** If the restricted state becomes empty ( $\perp$ ), we prove that this branch is **unreachable** (dead code).

2. **Path Sensitivity (Distinguishing History)** Even with control sensitivity, merging states at join points can be destructive.

**The Merge Problem:** Imagine one path sets  $x = 1$  and another sets  $x = -1$ . If we merge these into an interval  $[-1, 1]$  and then compute  $x * x$ , we get  $[0, 1]$ . But strictly speaking, on *both* original paths,  $x * x$  was exactly 1. The merge lost the specific values.

- **Solution:** Path sensitivity maintains separate abstract states for different paths reaching the same program point, rather than joining them immediately.
- **Trade-off:** This prevents “pollution” from imprecise paths but can lead to an exponential explosion in the number of states tracked.

3. **Relational Analysis (Tracking Dependencies)** Standard analyses (like Interval Analysis) are **non-relational** (or “Independent Attribute”). They track invariants for each variable in isolation (e.g.,  $x \in [0, 10]$  and  $y \in [0, 10]$ ).

**The Limitation:** If the code is ‘if  $(x > y)$ ’, a non-relational analysis cannot determine if this is true or false because it doesn’t know the *relationship* between  $x$  and  $y$ .

- **Solution:** Relational Analysis tracks constraints *between* variables, such as  $x = y$  or  $x < y + 5$ .
- **Example:** If we know  $x = y$ , we can prove ‘ $x > y$ ’ is impossible, identifying it as dead code—something impossible if we only looked at their ranges individually.