

Oral Exam Script: Static Program Analysis

Candidate

January 13, 2026

Topic 1: Type Analysis (Constraint-Based)

1. The Elevator Pitch

Definition: Type Analysis for TIP is a constraint-based static analysis that infers the types of variables and expressions in a program without executing it. Since TIP is not explicitly typed, we infer types to ensure consistency.

Why do we need it? We need it to guarantee **Type Safety**. We want to prevent runtime errors such as applying a non-function as a function, dereferencing an integer, or performing arithmetic on pointers.

Intuition: Imagine the AST as a puzzle. Each node has a "shape" (its type). We define rules for how pieces fit together (e.g., a dereference ' $*p$ ' implies ' p ' must be a pointer shape). If we can fit all pieces together without force, the program is well-typed.

2. The Technical Core (Constraint Generation)

[Note: This uses the Constraint-Based template.]

The Type Lattice/Terms: Our terms τ include `int`, pointers $\uparrow \tau$, functions $(\tau_1 \dots) \rightarrow \tau$, and recursive types $\mu \alpha. \tau$.

Constraint Generation: We assign a type variable $\llbracket E \rrbracket$ to every expression node E . We generate equality constraints based on syntax:

- **Allocations:** For ‘alloc E‘, the result is a pointer to the type of E:

$$\llbracket \text{alloc } E \rrbracket = \uparrow \llbracket E \rrbracket$$

- **Dereference:** For ‘*E‘, E must be a pointer to the result:

$$\llbracket E \rrbracket = \uparrow \llbracket *E \rrbracket$$

- **Functions:** For a call $E(A_1)$, E must be a function taking A_1 and returning the result:

$$\llbracket E \rrbracket = (\llbracket A_1 \rrbracket) \rightarrow \llbracket E(A_1) \rrbracket$$

The Solving Mechanism (Unification): We solve these constraints using the **Union-Find** algorithm (Unification). 1. We iterate through constraints $t_1 = t_2$. 2. We find the canonical representatives of t_1 and t_2 . 3. We **unify** them. If one is a type variable and the other is a proper type (like `int`), the variable adopts the proper type. 4. If we unify two incompatible proper types (e.g., `int` and $\uparrow \tau$), the program is **ill-typed**.

3. The Trace

Code Snippet:

```
var p, q;
p = alloc 10;
q = *p;
```

Step-by-Step Trace:

1. Generate Constraints:

- From ‘alloc 10‘: $\llbracket \text{alloc } 10 \rrbracket = \uparrow \llbracket 10 \rrbracket$. We know $\llbracket 10 \rrbracket = \text{int}$, so $\llbracket \text{alloc } 10 \rrbracket = \uparrow \text{int}$.
- From ‘p = ...‘: $\llbracket p \rrbracket = \llbracket \text{alloc } 10 \rrbracket$. Therefore, $\llbracket p \rrbracket = \uparrow \text{int}$.
- From ‘*p‘: $\llbracket p \rrbracket = \uparrow \llbracket *p \rrbracket$.
- From ‘q = ...‘: $\llbracket q \rrbracket = \llbracket *p \rrbracket$.

2. Solve (Unification):

- We know $\llbracket p \rrbracket = \uparrow \text{int}$.
 - Substitute into the dereference constraint: $\uparrow \text{int} = \uparrow \llbracket *p \rrbracket$.
 - Unify the inner types: $\text{int} = \llbracket *p \rrbracket$.
 - Substitute into q : $\llbracket q \rrbracket = \text{int}$.
3. **Result:** Consistent. p is a pointer to int, q is an int.

4. The Oral Exam Corner

The Hook: Recursive Types "A fascinating detail is how we handle cycles during unification, like $X = \uparrow X$. Standard unification would fail (occurs check), but in TIP, we allow **Regular Types**. We introduce a recursive binder μ , so the solution becomes $\mu\alpha. \uparrow \alpha$. This represents an infinite tree, allowing linked lists or self-referential structures."

The Trap: Polymorphism Examiner: "Does this analysis allow a function ‘id’ to be used for both integers and pointers?" **Answer:** "No, the standard analysis is **Monomorphic**. The function body is analyzed once. If ‘id’ maps $\alpha \rightarrow \alpha$, and we unify α with int at the first call, a second call with a pointer will fail. To fix this, we would need **Let-Polymorphism**, which instantiates fresh type variables for every usage of the function."

Topic 2: Monotone Frameworks (Dataflow)

1. The Elevator Pitch

Definition: The Monotone Framework is a general mathematical theory for dataflow analysis. Instead of creating ad-hoc algorithms for every analysis (Liveness, Reaching Definitions), we define a standard structure based on Lattices and Fixed Points.

Why do we need it? It provides a **guarantee of termination** and specific precision properties (Soundness) for any analysis that fits the framework constraints (monotonicity and finite height).

Intuition: Think of water flowing through a pipe network (the Control Flow Graph). The "water" is the information (e.g., which variables are live). The "pipes" are the instructions that might filter or add impurities (Transfer Functions). The "joints" are where pipes meet and water mixes (Join Operator).

2. The Technical Core (Monotone Framework)

[Note: This uses the Lattice/Dataflow template.]

1. The Lattice (L, \sqsubseteq): We define a complete lattice L representing abstract states.

- \perp (Bottom): Usually represents "no information" or "unreachable".
- \sqsubseteq (Partial Order): Represents precision. $x \sqsubseteq y$ means x is more precise (or safe) than y .

2. The Join Operator (\sqcup): This combines information from predecessor nodes (in forward analysis).

$$\text{Input}(v) = \bigsqcup_{u \in \text{pred}(v)} \text{Output}(u)$$

If we want "Must" properties (Intersections), \sqcup is \cap . If we want "May" properties (Unions), \sqcup is \cup .

3. The Transfer Functions f_ℓ : For every statement ℓ , we define $f_\ell : L \rightarrow L$. Typically defined as: $f_\ell(x) = (x \setminus \text{KILL}) \cup \text{GEN}$. **Constraint:** f must be **monotone**. If we know *less* about the input, we should know *less* (or the same) about the output.

3. The Trace (Sign Analysis)

Code Snippet:

```
var x;  
x = 0;      // Line 1  
x = x + 1; // Line 2
```

Lattice: The Sign Lattice $\{\perp, -, 0, +, \top\}$. **Trace:**

1. **Init:** $x \mapsto \top$ everywhere (or \perp if we optimize). Let's assume input to Line 1 is \top .
2. **Line 1 ($x = 0$):** Transfer function f_1 sets x to 0.

After Line 1 : $\{x \mapsto 0\}$

3. **Line 2 ($x = x + 1$):** Input is $\{x \mapsto 0\}$. Transfer function for $+$ looks at abstract table: $0 \oplus + = +$.

After Line 2 : $\{x \mapsto +\}$

4. **Fixed Point:** Further iterations do not change the state. We proved x is positive.

4. The Oral Exam Corner

The Hook: Widening "The standard framework relies on the lattice having **Finite Height** to guarantee the worklist algorithm terminates. However, powerful analyses like **Interval Analysis** have infinite height lattices. To solve this, we use **Widening** (∇). If we see a bound growing unstable ($[0, 1] \rightarrow [0, 2]$), we extrapolate immediately to infinity ($[0, \infty]$) to force termination, at the cost of precision."

The Trap: MOP vs. MFP Examiner: "Does this algorithm give the perfect answer?"
Answer: "Not necessarily. We compute the **Maximal Fixed Point (MFP)**. Ideally, we want the **Meet Over all Paths (MOP)**, which tracks every specific path. However, by Distributivity, if the transfer functions are distributive, MFP = MOP. If not (like in Constant Propagation), MFP is less precise but safe."

Topic 3: Control Flow Analysis (CFA)

1. The Elevator Pitch

Definition: Control Flow Analysis (CFA) determines the control flow graph for programs where the call targets are not known statically—specifically in the presence of **Function Pointers** (higher-order functions) or **Dynamic Dispatch**.

Why do we need it? Standard dataflow analysis assumes the CFG is already built. In TIP (or JS/Python), ‘`x()`’ could call anything. We cannot build the CFG without dataflow (knowing what ‘`x`’ is), and we can’t do dataflow without the CFG. CFA solves both simultaneously.

Intuition: It is a ”Chicken and Egg” problem. We model it as a flow graph where ”tokens” (functions) flow into variables. When a function token flows into a ”call site” bucket, we dynamically wire up the cables (edges) to that function’s body.

2. The Technical Core (The Cubic Framework)

[Note: This uses the Constraint-Based template.]

Constraint Generation: We define tokens $\{id\}$ and subset constraints \subseteq .

1. **Simple Flow:** For $x = y$, we generate $\llbracket y \rrbracket \subseteq \llbracket x \rrbracket$.
2. **Function Creation:** For ‘`foo() ...`’, we generate $\{foo\} \subseteq \llbracket foo \rrbracket$.
3. **Conditional Constraints (The Key):** For a call site $c : E(A)$, we generate:

$$\forall f : f \in \llbracket E \rrbracket \implies (\llbracket A \rrbracket \subseteq \llbracket f_{arg} \rrbracket \wedge \llbracket f_{ret} \rrbracket \subseteq \llbracket c \rrbracket)$$

The Solving Mechanism (Graph Reachability): We use a graph where nodes are AST variables and edges are inclusions. This is called the **Cubic Framework** ($O(n^3)$). Algorithm: 1. Propagate tokens along existing edges. 2. If a token f reaches a call site node $\llbracket E \rrbracket$, **add new edges** corresponding to the conditional constraint (Argument \rightarrow Parameter, Return \rightarrow Call). 3. Repeat until stable.

3. The Trace

Code Snippet:

```
var id, x;
id = (y) -> { return y; }; // Function f1
x = id(5);                // Call Site c1
```

Step-by-Step Trace:

1. Initial Setup:

- Function token $\{f1\}$ is created.
- Assignment: $\{f1\} \subseteq \llbracket id \rrbracket$.

2. Propagation:

- Token $\{f1\}$ flows into $\llbracket id \rrbracket$.

3. Conditional Trigger (at c1):

- Solver sees call ‘`id(5)`’. Checks $\llbracket id \rrbracket$.
- Finds $\{f1\}$. Trigger fires!
- **Add Edge (Args):** $\llbracket 5 \rrbracket \subseteq \llbracket f1_y \rrbracket$. (Abstract value of 5 flows to param y).
- **Add Edge (Ret):** $\llbracket f1_{ret} \rrbracket \subseteq \llbracket x \rrbracket$.

4. Final Flow:

- 5 flows to y . y returns to $f1_{ret}$. $f1_{ret}$ flows to x .
- Result: $\llbracket x \rrbracket$ contains 5 (or Int).

4. The Oral Exam Corner

The Hook: Context Sensitivity “The basic analysis I described is **0-CFA** (Context Insensitive). It merges all calls to a function. If we call ‘`id`’ with an Int and later with a Pointer, 0-CFA says the parameter can be *both*, and the return can be *both*, causing spurious flows (Int flowing to the pointer call site). **k-CFA** solves this by tagging tokens with the last k call sites (the call string), effectively separating the dataflow for different contexts.”

The Trap: Complexity Examiner: "You mentioned Cubic Time. Why is it $O(n^3)$?"

Answer: "In the constraint graph, we have variables (nodes) and subset relations (edges). In the worst case, every variable points to every other variable (n^2 edges), and every function flows across every edge (n functions). $n \times n^2 = O(n^3)$. This bottleneck is why scalable pointer analysis remains a significant challenge."