Type Analysis

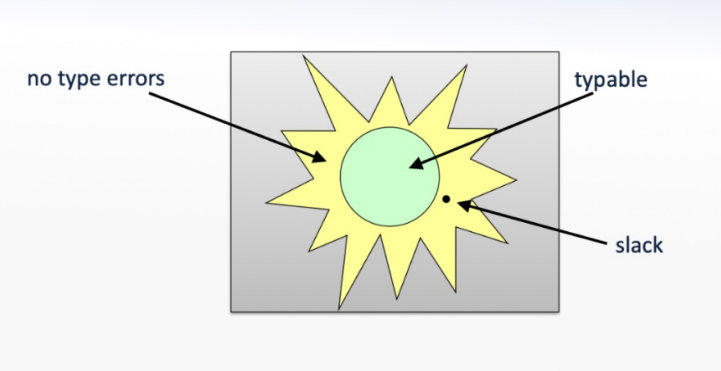
Introduction (2 Minutes)

Today we are discussing **Type Analysis** for the TIP language.

As we know from previous chapters, TIP does not have explicit type declarations. However, we implicitly expect operations to be reasonable. For example, we shouldn't try to add an integer to a function, or dereference a number.

- **Goal:** We want to ensure that these "reasonable restrictions" hold during execution to prevent runtime errors.
- **The Problem:** As established in Chapter 1, checking this perfectly is **undecidable**.
- **The Solution:** We use a **conservative approximation**. We define a program as *typable* if it satisfies a specific set of constraints.

Write on Whiteboard: Draw the "Sun" diagram (from the slides).



- *Inner Circle (Green):* Typable programs.
- *Outer Star (Yellow):* Programs with no runtime errors.
- *The Gap:* This is called **Slack**. Some valid programs will be rejected, but no invalid program will be accepted.

1. The Language of Types (3 Minutes)

To perform this analysis, we first need a formal language to describe the values in our program.

Write on Whiteboard:

$$\text{Type} \to \text{int} \mid \uparrow \text{Type} \mid (\text{Type}, \dots, \text{Type}) \to \text{Type}$$

We define types recursively:

1. **int**: Integers.
2. $\uparrow \tau$: Pointers to a type τ (represented by the up-arrow).
3. $(\tau_1, \dots) \to \tau_r$: Functions taking arguments and returning a type.

However, we have a problem. TIP allows recursive functions and data structures. A finite tree grammar isn't enough. We need **Regular Types** (essentially infinite trees that repeat).

To express this notationally, we add type variables (α) and a recursion operator (μ):

Write on Whiteboard:

$$\mu\alpha.(\uparrow \text{int}, \alpha) \to \text{int}$$

This represents a function that takes a pointer to an int and... *itself* as arguments.

2. Generating Type Constraints (5 Minutes)

This is the core of the analysis. We traverse the program's Abstract Syntax Tree (AST) and generate equality constraints for every node.

For every identifier X and every expression node E, we assign a type variable, denoted by double brackets: $[\![X]\!]$ or $[\![E]\!]$.

Let's look at the specific rules.

Write on Whiteboard - Rapidly list these rules as you explain:

A. Basic Operations

We must distinguish between general binary operators and the equality operator, as they generate different constraints.

1. Arithmetic and Inequalities (E_1 op E_2 where op $\in \{+, -, >, \dots\}$): For most operators, we demand that both operands are integers and the result is an integer.

$$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \text{ op } E_2]\!] = \text{int}$$

2. Equality Checks ($E_1 == E_2$): The equality operator is stricter about the result (it must be a boolean/integer condition), but more flexible about the inputs. The

operands E_1 and E_2 do not have to be integers; they simply must unify with *each other* (e.g., we can compare two pointers).

$$[\![E_1]\!] = [\![E_2]\!] \quad \wedge \quad [\![E_1 == E_2]\!] = \text{int}$$

B. Pointers and Allocation

This is crucial.

- **Alloc:** `alloc E` creates a pointer.

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

- **Address of:** `&X` is also a pointer.

$$[\![\&X]\!] =\uparrow [\![X]\!]$$

- **Dereference:** `*E`. If we dereference E, then E itself must be a pointer to the result.

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

C. Functions

For a function definition $X(X_1, \ldots, X_n)\{\ldots \text{return } E;\}$:

$$[\![X]\!] = ([\![X_1]\!], \ldots, [\![X_n]\!]) \to [\![E]\!]$$

The type of the function identifier matches its signature constructed from the parameters and the return body expression.

D. The "Null" Problem

`null` is polymorphic—it can be a pointer to anything. We assign it a **fresh type variable** α every time it appears.

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

3. Solving Constraints: Unification (3 Minutes)

Once we generate these constraints, we have a system of equations like $[\![X]\!] = [\![Y]\!]$ or $[\![X]\!] =\uparrow \text{int}$. How do we solve them? We use the **Unification Algorithm**.

Speaker Notes: This is efficient (almost linear time) using the **Union-Find** data structure.

1. **Initialization:** We start by treating every type variable and term as being in its own set (invoking `MAKESET`).
2. **Processing Constraints:** For every equality $\tau_1 = \tau_2$, we use `FIND` to look up the **canonical representative** of each term's equivalence class.
3. **Merging:** We use `UNION` to merge the sets. Crucially, proper types (like `int` or function types) take precedence as representatives over variables.

4. **Type Errors:** If we attempt to `UNION` two sets that already have conflicting proper types as representatives (e.g., one represents `int` and the other represents $\uparrow \tau$), the algorithm reports a **Type Error**.

Example for Whiteboard: Consider the code:

```
p = alloc null;
*p = p;
```

Let's generate constraints:
1. $[\![\text{null}]\!] =\uparrow \alpha$
2. $[\![\text{alloc null}]\!] =\uparrow [\![\text{null}]\!] =\uparrow\uparrow \alpha$
3. $[\![p]\!] = [\![\text{alloc null}]\!]$
4. $[\![*p]\!] = [\![p]\!]$ (from assignment)
5. $[\![p]\!] =\uparrow [\![*p]\!]$ (from dereference usage)

The Recursive Step: Combining 4 and 5 yields:

$$[\![p]\!] =\uparrow [\![p]\!]$$

When the Union-Find algorithm processes this:

- It unifies the set containing $[\![p]\!]$ with the set containing the term $\uparrow [\![p]\!]$.
- The representative of $[\![p]\!]$ becomes the term $\uparrow \ldots$, which refers back to $[\![p]\!]$.

This successfully describes the infinite (regular) type:

$$[\![p]\!] = \mu\tau.\uparrow \tau$$

4. Records and Limitations (2 Minutes)

Records

We can extend types to include records: $\{id : \text{Type}, \ldots\}$. However, dealing with field access `E.x` is tricky because we don't know if the field exists just by looking at the syntax.

We model this by assuming *all* possible fields exist in the type, but some are marked as **absent** ($\bullet$).

$$[\![\{f : 1\}]\!] = \{f : \text{int}, g : \bullet, h : \bullet \ldots\}$$

If we try to access a field that resolves to $\bullet$, that is a type error.

Limitations

Finally, we must acknowledge the limitations of this analysis.

1. **Flow Insensitivity:**

```
var x;
x = alloc 1;  // x is pointer
x = 42;       // x is int
```

This is valid in TIP, but our analysis assigns *one* type variable $[\![x]\!]$ to x. It cannot be both `int` and `ptr`. The program is rejected (Slack).

2. **Polymorphism:** Our system is monomorphic. If we define a generic function `id(x) { return x; }`, we calculate its constraints once. If we call it with an `int`, constraints say it returns `int`. If we later call it with a pointer, we get a unification error. *(Note: We could use Let-Polymorphism to fix this, but it makes complexity exponential).*

Conclusion

To wrap up: We have built a static analysis that guarantees the absence of type errors. It is efficient, using Union-Find, but it is conservative—it will reject some valid programs (slack) due to flow-insensitivity and lack of polymorphism.