Assignment #2: Typed Lambda Calculus

Overview

The goal of this assignment is to build the foundational type system implementation over lambda expressions with simple boolean and arithmetic terms.

Starting Point

Copy the enumerated type for `Expression` from Assignment #1 declared in `src/arith/expr.rs`. You can also copy the utility functions in `src/arith/build.rs` if you wish.

You will eventually need to extend the definition of `Expression` to include variables, function applications, and function abstractions. You can do this immediately, or after implementing the typing rules for boolean and arithmetic expressions.

Representing Types

Define a new enumerated type to represent the valid types for this language. As you can infer from the typing rules on the last page, the valid types are `Bool`, `Int`, and function types.

Typing Context

You will need to determine a representation for a typing context (i.e., Γ). There are many perfectly valid representations. For the purposes of the course project, you might consider a vector or a `HashMap`.

Whichever data structure you choose, give careful consideration to how mutation of the data structure might affect your implementation.

Tests

Include in your submission a set of tests verifying that your implementation works as expected, both for terms that are well-typed and terms that are not.

At a minimum, include a set of tests translated from Problems 1 and 3 in the Ungraded Problem Set #2.

Typing Function

Implement, in Rust, the typing rules given on the last page.

As with the evaluator in Assignment #1, your type checker is to be implemented as a function (with additional supporting functions) in a library (i.e., there is no need for a main; this function will be exercised through test cases). This function should take a term to check and a typing context as arguments. This function should return a `Result` of either the (`Ok`) type for the term or an (`Err`) error string indicating the typing rule that failed (as labeled in the set of typing rules) and the reason for the failure. You have some freedom to choose how these errors will look (and, if you prefer, you can use a type other than string for the error), but they should provide adequate information about what went wrong.

Grading

Grading will be divided as follows, and will be based on both functionality and quality of implementation.

Part	Percentage
Bool (T-True, T-False, T-IF)	15
Int (T-IntConst, T-IsZero, T-Add, T-Sub)	15
Lambda (T-Var, T-Abs, T-App)	60
Tests	10

The terms for the expression language are to be inferred from the rules below coupled with the discussion in lecture (and in the textbook). Your implementation, of course, will work on the internal AST representation of such expressions, so you should be able to map the terms used in the typing rules to the variants declared in the Rust code.

nv is for numeric values.

$$\Gamma \vdash \text{true} : \text{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \text{false} : \text{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \tau} \qquad \text{(T-If)}$$

$$\Gamma \vdash nv : \text{Int} \qquad \text{(T-IntConst)}$$

$$\frac{t_1 : Int}{\Gamma \vdash \text{iszero } t_1 : \text{Bool}} \qquad \text{(T-IsZero)}$$

$$\frac{\Gamma \vdash t_1 : \text{Int} \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash t_1 + t_2 : \text{Int}} \qquad \text{(T-Add)}$$

$$\frac{\Gamma \vdash t_1 : \text{Int} \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash t_1 - t_2 : \text{Int}} \qquad \text{(T-Sub)}$$

$$\frac{\text{x} : \tau \in \Gamma}{\Gamma \vdash \text{x} : \tau} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \text{x} : \alpha \vdash t_1 : \beta}{\Gamma \vdash \lambda \text{x} : \alpha.\ t_1 : \alpha \to \beta} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \alpha \to \beta \quad \Gamma \vdash t_2 : \alpha}{\Gamma \vdash t_1\ t_2 : \beta} \qquad \text{(T-App)}$$