TYPED LAMBDA CALCULUS

Like the untyped lambda calculus is the foundation for untyped languages, the simply typed lambda calculus is the foundation for typed (functional) languages.

GRAMMAR:-

$$
\begin{aligned}
M &\longrightarrow x \\
&\mid \lambda x : T . M \qquad \leftarrow \text{Type} \\
&\mid M_1 \; M_2
\end{aligned}
$$

As we may have nested functions, types are usually defined relative to a "type environment" (Γ), which basically binds variables to types.

EVALUATION:-

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2} \; (1) \qquad \frac{t_2 \longrightarrow t_2'}{v_1 t_2 \longrightarrow v_1 t_2'} \; (1)$$

Just tells that operators are evaluated before operands.

TYPING:-

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T} \text{ (var)} \qquad \frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1 . t_2 : T_1 \to T_2} \text{ (abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma : t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ (app)}$$

∴ will fail if t_1 is not a function (i.e., arrow type)

$\vdash$ means in the type environment.

, extends a type environment.

In $\Gamma \vdash t:T$, Γ is a set of type bindings for the free variables in T.

TYPE SOUNDNESS THEOREM

Well-typed programs cannot "go wrong".

– Robin Milner (1978)

Defined by ① Progress ; ② Preservation.

① If $\vdash t:T$ for some type T, then either t is a value or $\exists t', t \to t'$.

② If $\Gamma \vdash t:T$ and $t \to t'$, then $\Gamma \vdash t':T$.

evaluation pre... well-typedness.

If we can't make progress, we are stuck ⇒ type error.

EXAMPLE:-

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

Now, say we got an expression "if true then 0 else "hello"".

This won't type check in our system.

We can extend type systems to support features:

SUBTYPING

$S \leq T$ means S is a "subtype" of T.

i.e., it may have more properties than T!

"Sub" because elements of S would be a subset of that of T, as S has stricter constraints than T.

Subtyping follows "Principle of safe substitution":-

$$\frac{\Gamma \vdash t : S \qquad S \leq T}{\Gamma \vdash t : T}$$

i.e., every element of S is also an element of T.

Two rules :-

$S \leq S$ (reflexive) $\qquad \dfrac{S \leq U \qquad U \leq T}{S \leq T}$ (transitive)

For convenience: $S \leq Top$, where Top is a supertype of every type (transitivity helps).

Subtyping for functions :-

$$\frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

Subtyping for function (arrow) types is contravariant for its argument types, but covariant for the result type.

→ A subtype can be supplied for an argument, and a supertype can hold the result.