The following derivation shows that
\[ \lambda x. \lambda y. (x y) \] can be assigned the type \( (T_1 \rightarrow T_2) \rightarrow (T_1 \rightarrow T_2) \), for any types \( T_1 \) and \( T_2 \):

\[
\begin{align*}
\Gamma & \vdash x : T_1 \\
\Gamma & \vdash t_1 : T_2 \rightarrow T_1 \\
\Gamma & \vdash t_2 : T_2 \\
\Gamma, x : T_1 & \vdash t : T_2 \\
\Gamma & \vdash \lambda x. t : T_1 \rightarrow T_2
\end{align*}
\]

An important property of our type assignment system, which we will prove next time, is the following:

**Theorem 1 (Type Preservation)** If \( \Gamma \vdash t : T \) and \( t \rightarrow_\beta t' \) (full \( \beta \)-reduction), then \( \Gamma \vdash t' : T \).

**Algorithmic typing.** We can try to use the rules algorithmically by starting with some goal type assignment to prove, and matching the conclusion of a rule to that goal. The appropriately instantiated premises then become the new goals, and we proceed recursively. If you are familiar with logic programming as in Prolog, this is a similar idea. There are two ways we might try to use these rules in this way, depending on which of \( \Gamma \), \( t \), and \( T \) we consider to be inputs, and which outputs. Unfortunately, both of which end up being infinitarily non-deterministic (and hence unusable). So we will have to refine the rules in some way to get a deterministic algorithm.

1. **Type checking.** On this approach, we take \( \Gamma \), \( t \), and \( T \) as inputs (and there are no outputs). So the judgment expresses that we check whether \( t \) can be assigned simple type \( T \) in context \( \Gamma \). The problem with this reading is that when we apply the application rule, we must non-deterministically guess type \( T_2 \) as we pass from its conclusion to its premises. There are an infinite number of choices, since there are infinitely many simple types. Note, however, that the other rules can both be executed deterministically.
2. **Type computation.** We can also take $\Gamma$ and $t$ as inputs, and $T$ as output. In this case, the judgment expresses the idea that simple type $T$ can be computed for $t$ in context $\Gamma$. The application rule is completely deterministic on this reading: if we have computed type $T_2 \rightarrow T_1$ for $t_1$ and type $T_2$ for $t_2$, then we compute type $T_1$ for the application of $t_1$ to $t_2$. The problem with the type computation reading shows up in the rule for typing $\lambda$-abstractions. There, we must non-deterministically guess the type $T_1$ to give to $x$ in the extended context in the premise of the rule. So once again, the rules are infinitarily non-deterministic.

We now consider different ways to obtain a deterministic algorithm for typing:

1. **Annotated applications for type checking.** If we wish to use the typing rules for type checking, then we can annotate applications to remove the non-determinism in the application rule (described above). We may add annotations to applications by the syntax of $\lambda$-terms $t$:

   $t ::= x \mid (t_1 \ t_2) \mid [T] \mid \lambda x.t$

   The typing rules above are then modified as follows (note that only the application rule has changed):

   $\Gamma(x) = T \quad \Gamma \vdash t_1 : T_2 \rightarrow T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash \lambda x. t : T_1 \rightarrow T_2$

   This approach is admittedly not commonly used in practice, though it is theoretically sufficient.

2. **Annotated abstractions for type computation.** More commonly, if we wish to use the typing rules for type computation, then we can annotate $\lambda$-abstractions to remove the non-determinism in the abstraction rule:

   $t ::= x \mid (t_1 \ t_2) \mid \lambda x : T. t$

   The typing rules above are then modified as follows (only the $\lambda$-abstraction rule has changed):

   $\Gamma(x) = T \quad \Gamma \vdash t_1 : T_2 \rightarrow T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash \lambda x : T_1 \rightarrow T_2$

3. **Constraint generation.** Rather than add annotations, we modify our typing rules so they generate constraints. This idea can be implemented based on an interpretation of the typing judgment as expressing type checking, as well as on an interpretation as type computation. Here, we pursue the latter. The rules now operate on judgments of the form $\Gamma \vdash t : T > C$, where $\Gamma$ and $t$ are inputs, and $T$ and $C$ are outputs. $C$ is a set of constraints which must be satisfied in order for the type assignment to hold. A constraint is an equation between simple types with meta-variables $X$ defined by the following syntax:

   $T ::= b \mid X \mid T_1 \rightarrow T_2$

   The constraint generation rules are the following (where $\cdot$ denotes the empty set of constraints, and comma is used for unioning sets of constraints). In the rule for $\lambda$-abstractions and the rule for applications, $X$ is a new meta-variable (not occurring in any other term, type, or context listed).

   $\Gamma(x) = T \quad \Gamma \vdash t_1 : T_1 \rightarrow C_1 \quad \Gamma \vdash t_2 : T_2 \rightarrow C_2 \quad \Gamma \vdash x : T \rightarrow C \quad \Gamma, x : X \vdash t : T > C \quad \Gamma \vdash \lambda x. t : X \rightarrow T > C$

**Next time:** type preservation, constraint solving using unification.