Introduction

Dependent types are an extension to traditional types (such as the one seen in the simply-typed lambda calculus) to make them more expressive. Specifically, a type system is said to be dependent if it allows types to depend on terms.

This type theory is based on Per Martin-Löf's original type theory, which was posited as an alternative foundation for mathematics. In it, we can encode any constructive mathematical theorem.

In this assignment, you'll be implementing a small language which uses dependent types, building up the language to include new features, then ultimately proving some simple theorems using the language. The next few sections define the language formally, and it will be your task to implement the language yourself.

Syntax

$$
\begin{array}{rll}
e, \tau ::= & x & \text{Variable reference} \\
| & \star & \text{The type of types} \\
| & (x : \tau_1) \to \tau_2 & \text{Dependent function types (AKA } \Pi\text{-types)} \\
| & \lambda(x : \tau_1).\ e_2 & \text{Function introduction} \\
| & e_1\ e_2 & \text{Function elimination} \\
| & \mathbb{N} & \text{The natural number type} \\
| & 0 \mid \texttt{succ}\ e & \text{Natural number introduction} \\
| & \texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 & \text{Natural number elimination}
\end{array}
$$

A few bits of syntactic sugar:

- We occasionally use the name '_' to bind values we don't care about. This name should not appear in the variable reference position.
- We can elide names in Π-types if they are not bound in the type's codomain: $e_1 \to e_2 \equiv (\_ : e_1) \to e_2$.
- We write the function arrow as right-associative: $(x : e_1) \to (x : e_2) \to e_3 \equiv (x : e_1) \to ((x : e_2) \to e_3)$.
- We write function elimination as left associative: $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$.
- We write a natural number as the repeated application of `succ` to `0`. As an example, we say that $4 \equiv \texttt{succ}(\texttt{succ}(\texttt{succ}(\texttt{succ}\ 0)))$.

Note: α-equivalence

In the above syntax, we use the non-terminal x to stand for any variable. In general, the particular variable names we choose for a closed term should not matter. For example, $\lambda(x : \mathbb{N}).x \equiv \lambda(y : \mathbb{N}).y$. This property — that consistently renaming a parameter and all of its occurrences does not change the meaning of a term — is called α-equivalence. In the following sections, we often make the implicit assumption that terms are renamed to avoid collision.

Meta-Functions

Environments

Type-checking is performed relative to a **environment**, also called a context, which is often named Γ. An environment is a list of type/variable pairs, written $(x_1 : \tau_1, x_2 : \tau_2, \ldots)$. We write $\Gamma(x)$ to mean the type which is most recently associated with the variable x in the environment Γ. Note: this is not necessarily defined for all variables.

$$\Gamma(x) = \begin{cases} \tau & \text{if } \Gamma = (\ldots, x : \tau) \\ (\ldots)(x) & \text{if } \Gamma = (\ldots, x_1 : \tau) \text{ and } x_1 \neq x \end{cases}$$

Free Variables

We say that a variable is **free** in some term when that variable occurs in the term, but is not bound by a binding term (such as a λ or Π term). We write the set of free variables present in a term e as $\mathcal{FV}(e)$.

$$\mathcal{FV}(e) = \begin{cases} \{x\} & \text{if } e = x \\ \mathcal{FV}(\tau_1) \cup (\mathcal{FV}(\tau_2) - \{x\}) & \text{if } e = (x : \tau_1) \to \tau_2 \\ \mathcal{FV}(\tau_1) \cup (\mathcal{FV}(e_1) - \{x\}) & \text{if } e = \lambda(x : \tau_1).e_1 \\ \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) & \text{if } e = e_1\ e_2 \\ \mathcal{FV}(e_1) & \text{if } e = \texttt{succ}\ e_1 \\ \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \cup \mathcal{FV}(e_3) \cup \mathcal{FV}(e_4) & \text{if } e = \texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 \\ \emptyset & \text{otherwise} \end{cases}$$

Capture-Avoiding Substitution

When evaluating application, we often use a notion of **substitution** - we replace all occurrences of the argument variable with the argument value. This is a suitable mental model for many programs, but a naive notion of substitution can lead to odd bugs. We write $e_1[x \hookleftarrow e_2]$ to mean "the term e_1, but with all free occurrences of x replaced with e_2". Note: $e_1[x \hookleftarrow e_2]$ is not necessarily defined for all terms - some may require rewriting in terms of α-equivalence.

$$e_1[x \hookleftarrow e_2] = \begin{cases} e_2 & \text{if } e_1 = x \\ (x_1 : \tau_1[x \hookleftarrow e_2]) \to \tau_2[x \hookleftarrow e_2] & \text{if } e_1 = (x_1 : \tau_1) \to \tau_2 \\ & \text{ and } x_1 \notin \{x\} \cup \mathcal{FV}(e_2) \\ \lambda(x_1 : \tau_1[x \hookleftarrow e_2]).e_3[x \hookleftarrow e_2] & \text{if } e_1 = \lambda(x_1 : \tau_1).e_3 \\ & \text{ and } x_1 \notin \{x\} \cup \mathcal{FV}(e_2) \\ e_3[x \hookleftarrow e_2]\ e_4[x \hookleftarrow e_2] & \text{if } e_1 = e_3\ e_4 \\ \texttt{succ}\ e_3[x \hookleftarrow e_2] & \text{if } e_1 = \texttt{succ}\ e_3 \\ \texttt{elimNat}\ e_3[x \hookleftarrow e_2]\ e_4[x \hookleftarrow e_2]\ e_5[x \hookleftarrow e_2]\ e_6[x \hookleftarrow e_2] & \text{if } e_1 = \texttt{elimNat}\ e_3\ e_4\ e_5\ e_6 \\ e_1 & \text{otherwise} \end{cases}$$

Semantics

There are two relevant judgments, $\Gamma \vdash e : \tau$ (read as "environment Γ types e as τ") and $e_1 \rightsquigarrow e_2$ (read as "e_1, after one step of evaluation, gives e_2"). The relation $e_1 \rightsquigarrow^* e_2$ is the repeated application of evaluation rules until no more apply.

Typing Relation

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Type-Var-Ref}
\qquad
\frac{}{\Gamma \vdash \star : \star}\ \textsc{Type-}\star
\qquad
\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 : \star}{\Gamma \vdash (x : \tau_1) \to \tau_2 : \star}\ \textsc{Type-}\Pi
$$

$$
\frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash \tau_2 : \star \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e_2 : (x : \tau_1) \to \tau_2}\ \textsc{Type-}\lambda
\qquad
\frac{\tau_1 \rightsquigarrow^* \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}\ \textsc{Type-Eval}
$$

$$
\frac{\Gamma \vdash e_1 : (x : \tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2[x \leftarrow e_2]}\ \textsc{Type-App}
$$

$$
\frac{}{\Gamma \vdash \mathbb{N} : \star}\ \textsc{Type-}\mathbb{N}
\qquad
\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ \textsc{Type-0}
\qquad
\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \texttt{succ}\ e : \mathbb{N}}\ \textsc{Type-}\texttt{succ}
$$

$$
\frac{\Gamma \vdash e_1 : \mathbb{N} \to \star \qquad \Gamma \vdash e_2 : e_1\ 0 \qquad \Gamma \vdash e_3 : (x : \mathbb{N}) \to e_1\ x \to e_1\ (\texttt{succ}\ x) \qquad \Gamma \vdash e_4 : \mathbb{N}}{\Gamma \vdash \texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 : e_1\ e_4}\ \textsc{Type-}\texttt{elimNat}
$$

Example: Polymorphic Identity Function

The following is a derivation which types the polymorphic identity function `id`, defined as follows:

$$
\begin{aligned}
\texttt{id} &: (A : \star) \to A \to A \\
\texttt{id} &= \lambda(A : \star).\lambda(x : A).x
\end{aligned}
$$

We can prove that this type is accurate using a derivation:

$$
\cfrac{
\begin{array}{c}
\cfrac{}{\vdash \star : \star}\ \textsc{Type-}\star \\
\cfrac{\cfrac{(A : \star)(A) = \star}{A : \star \vdash A : \star}\ \textsc{Type-Var-Ref} \qquad \cfrac{(A : \star, \_ : A)(A) = \star}{A : \star, \_ : A \vdash A : \star}\ \textsc{Type-Var-Ref}}{A : \star \vdash A \to A : \star}\ \textsc{Type-}\Pi \\
\cfrac{
\begin{array}{c}
\cfrac{(A : \star)(A) = \star}{A : \star \vdash A : \star}\ \textsc{Type-Var-Ref} \\
\cfrac{(A : \star, x : A)(A) = \star}{A : \star, x : A \vdash A : \star}\ \textsc{Type-Var-Ref} \qquad \cfrac{(A : \star, x : A)(x) = A}{A : \star, x : A \vdash x : A}\ \textsc{Type-Var-Ref}
\end{array}
}{A : \star \vdash \lambda(x : A).x : A \to A}\ \textsc{Type-}\lambda
\end{array}
}{\vdash \lambda(A : \star).\ \lambda(x : A).\ x : (A : \star) \to A \to A}\ \textsc{Type-}\lambda
$$

Evaluation Relation

The following defines the evaluation relation. This may seem like a lot, but only the first three do any interesting work (these are called "computational rules"). The rest just thread through evaluation to various subterms (these are called "congruence rules").

$$\frac{}{(\lambda(x:\tau_1).e_1)\ e_2 \rightsquigarrow e_1[x \hookleftarrow e_2]}\ \textsc{Eval-App} \qquad \frac{}{\texttt{elimNat}\ e_1\ e_2\ e_3\ 0 \rightsquigarrow e_2}\ \textsc{Eval-}\texttt{elimNat}\textsc{-}0$$

$$\frac{}{\texttt{elimNat}\ e_1\ e_2\ e_3\ (\texttt{succ}\ e_4) \rightsquigarrow e_3\ e_4\ (\texttt{elimNat}\ e_1\ e_2\ e_3\ e_4)}\ \textsc{Eval-}\texttt{elimNat-succ}$$

$$\frac{\tau_1 \rightsquigarrow \tau_2}{(x:\tau_1) \to \tau_3 \rightsquigarrow (x:\tau_2) \to \tau_3}\ \textsc{Eval-}\Pi\textsc{-domain} \qquad \frac{\tau_2 \rightsquigarrow \tau_3}{(x:\tau_1) \to \tau_2 \rightsquigarrow (x:\tau_1) \to \tau_3}\ \textsc{Eval-}\Pi\textsc{-codomain}$$

$$\frac{e_1 \rightsquigarrow e_2}{\lambda(x:\tau_1).e_1 \rightsquigarrow \lambda(x:\tau_1).e_2}\ \textsc{Eval-}\lambda\textsc{-Body} \qquad \frac{e_1 \rightsquigarrow e_2}{e_1\ e_3 \rightsquigarrow e_2\ e_3}\ \textsc{Eval-Rator} \qquad \frac{e_1 \rightsquigarrow e_2}{e_3\ e_1 \rightsquigarrow e_3\ e_2}\ \textsc{Eval-Rand}$$

$$\frac{e_1 \rightsquigarrow e_2}{\texttt{succ}\ e_1 \rightsquigarrow \texttt{succ}\ e_2}\ \textsc{Eval-}\texttt{succ} \qquad \frac{e_1 \rightsquigarrow e_5}{\texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 \rightsquigarrow \texttt{elimNat}\ e_5\ e_2\ e_3\ e_4}\ \textsc{Eval-}\texttt{elimNat}\textsc{-mot}$$

$$\frac{e_2 \rightsquigarrow e_5}{\texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 \rightsquigarrow \texttt{elimNat}\ e_1\ e_5\ e_3\ e_4}\ \textsc{Eval-}\texttt{elimNat}\textsc{-base}$$

$$\frac{e_3 \rightsquigarrow e_5}{\texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 \rightsquigarrow \texttt{elimNat}\ e_1\ e_2\ e_5\ e_4}\ \textsc{Eval-}\texttt{elimNat}\textsc{-ind}$$

$$\frac{e_4 \rightsquigarrow e_5}{\texttt{elimNat}\ e_1\ e_2\ e_3\ e_4 \rightsquigarrow \texttt{elimNat}\ e_1\ e_2\ e_3\ e_5}\ \textsc{Eval-}\texttt{elimNat}\textsc{-target}$$