1 Injection and discrimination

Consider the type of lists, defined in the prelude of Coq:

\[
\text{Inductive list (A:Type) :=}
\ni\text{nil} \mid \text{cons (\_:A) (\_:list A)}.
\]

Prove, without using the dedicated tactics \textit{injection} and \textit{discriminate}, the following properties on lists:

1. Injectivity of \textit{cons}: \text{cons x1 l1 = cons x2 l2} \rightarrow \text{x1 = x2} /\!\! \text{ l1 = l2}
2. Discrimination of \text{nil} and \text{cons}: \text{nil} \not\!\! \leftrightarrow \text{cons x l}

2 Strict positivity

2.1 Encoding pure \(\lambda\)-calculus in a non-positive inductive type

In this section we analyze the consequence of having an inductive type which does not comply with the strict positivity condition. We (partially) simulate the inductive definition

\[
\text{Inductive lambda := Lam (_:lambda->lambda)}.
\]

by assuming the introduction and case-analysis rules:

\[
\begin{align*}
\text{Parameter lambda : Type}. \\
\text{Parameter Lam : (lambda->lambda) \rightarrow lambda}. \\
\text{Parameter match_lambda : forall P:lambda -> Type, <todo> \rightarrow forall l, P l}. \\
\end{align*}
\]

The \(\iota\)-reduction is represented by an equation:

\[
\begin{align*}
\text{Parameter lambda_eq : forall P H f, match_lambda P H (Lam f) = H f}. \\
\end{align*}
\]

1- Encode the pure \(\lambda\)-calculus in this type, by defining application \textit{app} : lambda \rightarrow lambda \rightarrow lambda and the \(\beta\)-equality \textit{app (Lam f)x} = f x
2- Prove that any function \(f : lambda \rightarrow lambda\) has a fixpoint. That is, there exists a term \(t\) such that \(f t = t\).
3- Show that the above axiomatization of \textit{lambda} does not introduce an inconsistency by exhibiting a \textit{model}. Complete the following piece of theory:

\[
\begin{align*}
\text{Definition L : Type := <todo>}. \\
\text{Definition Li (f:L->L) : L := <todo>}. \\
\text{Definition Lm (P:L->Type) (H:<todo>) (l:L) : P l := <todo>}. \\
\text{Lemma L_eq P H f : Lm P H (Li f) = H f}. \\
\end{align*}
\]

4- Introduce a parameter \textit{rec_lambda} encoding this recursive scheme.
5- Show that it is inconsistent.
6- Explain why the following fixpoint is not accepted?
In this document, we have extracted a section on termination of fixpoints. The text contains several definitions and examples, including the following:

### Termination of fixpoints

Are the following fixpoints well-formed? Explain why?

```coq
Fixpoint leq (n p: nat) {struct n} : bool :=
  match n, p with
  | O, _ => true
  | S _, 0 => false
  | S n', S p' => leq n' p'
end.
```

Definition exp (p:nat) :=
  fix f (n:nat) : nat :=
  match leq p n with | true => S 0 | false => f (S n) + f (S n) end
0.

Definition ackermann1 := fix f (n m:nat) : nat :=
  match n, m with
  | O, _ => S m
  | S n', 0 => f n' (S 0)
  | S n', S m' => f n' (f n m')
end.

Definition ackermann2 := fix f (n:nat) : nat -> nat :=
  match n with
  | O => S
  | S n' => fix g (m:nat) : nat :=
    match m with
    | O => f n' (S 0)
    | S m' => f n' (g m')
    end
end.
```

### The type \( W \) of well-founded trees

The type \( W \) of well-founded trees is parameterised by a type \( A \) and a family of types \( B : A \to Type \). It has only one constructor and is defined by:

```coq
Inductive W (A:Type) (B:A -> Type) : Type :=
  node : forall (a:A), (B a -> W A B) -> W A B.
```

The type \( A \) is used to parameterise the nodes and the type \( B a \) give the arity of the node parameterised by \( a \).

1. Give the type of dependent elimination for type \( W \) on sort \( Type \).

2. In order to encode the type \( nat \) of natural numbers with \( O \) and \( S \), we need two types of nodes. We take \( A \equiv bool \).

   The constructor \( O \) corresponds to \( a \equiv false \), it does not expect any argument so we take \( B \equiv empty \). The constructor \( S \) corresponds to \( a \equiv true \), it takes one argument, we define \( B \equiv unit \).

   Using this encoding, give the terms corresponding to \( nat \), \( O \) et \( S \).

3. Propose an encoding using \( W \) for the type \( tree \) of binary trees parameterised by a type of values \( V \), which means that we have a constructor \( leaf \) of type \( tree V \) and a constructor \( bin \) of type \( tree V \to V \to tree V \to tree V+ \). Define the type and its constructors using this encoding.
4. Given a variable \( n \) of type \( \text{nat} \), build two functions \( f_1 \) and \( f_2 \) of type \( \text{unit} \rightarrow \text{nat} \) such that \( \forall x: \text{unit}, f_i \ x = n \) is provable but such that \( f_1 \) and \( f_2 \) are not convertible.

5. Which consequence does it have on the encoding of \( \text{nat} \) using \( W \)?