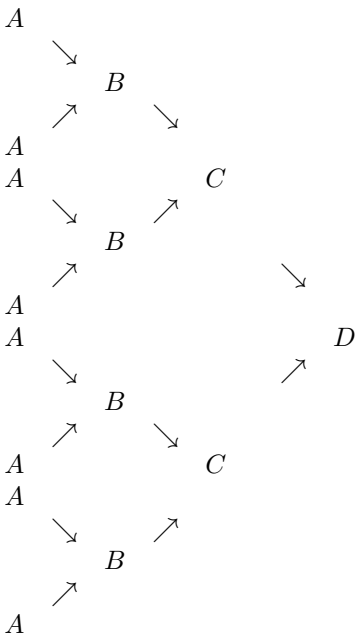
Homework

1. Identify exactly where in the translation *tree*, mapping a box-style proof to a tree-style proof, the exponential blow-up occurs, for the formulas $A \to (A \to A \to B) \to (B \to B \to C) \to (C \to C \to D) \to$ ... as presented in the PS.

   Say D is the conclusion to be proven. The box-style proof looks, qua usage of assumptions, like a dag (directed acyclic graph): $A \rightrightarrows B \rightrightarrows C \rightrightarrows D$. Translating to a tree usage of assumptions:



   (This correspondence is the same as the one between BDD and term representations of formulas.)

2. Install Coq

   `https://coq.inria.fr/download`

   and do the tutorial

   `https://coq.inria.fr/tutorial-nahas`

3. Prove $(A \to B \to C) \to (A \to B) \to (A \to C)$ in Coq in stepwise fashion (using intro and apply), and show in each step what is the goal, what are the assumptions, and what is the proof (in tree-style) constructed thus far (if you so wish, this proof may be represented by its λ-term instead).

```
Theorem exercise3 : (forall A B C : Prop, (A -> B -> C) -> (A -> B) -> A -> C).
Proof.
intros A B C.
intro A_impl_B_impl_C.
intro A_impl_B.
intro proof_of_A.
```

```
apply A_impl_B_impl_C.
exact proof_of_A.
apply A_impl_B.
exact proof_of_A.
Qed.
Print exercise3.
```

After the first line, we have:

$$\frac{\dots}{(A \to B \to C) \to (A \to B) \to A \to C}$$

then, for assumption `A_impl_B_impl_C`: $A \to B \to C$*:*

$$\frac{\dfrac{\dots}{(A \to B) \to A \to C}}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

then, for assumptions `A_impl_B_impl_C`: $A \to B \to C$, `A_impl_B`: $A \to B$*:*

$$\frac{\dfrac{\dfrac{\dots}{A \to C}}{(A \to B) \to A \to C} \to I}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

then, for assumptions $\Gamma =$`A_impl_B_impl_C`: $A \to B \to C$, `A_impl_B`: $A \to B$, `proof_of_A`: A*:*

$$\frac{\dfrac{\dfrac{\dfrac{\dots}{C}}{A \to C} \to I}{(A \to B) \to A \to C} \to I}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

then, for assumptions Γ*:*

$$\frac{\dfrac{\dfrac{[A \to B \to C] \quad \dfrac{\dots}{A}}{C} \to E \quad \dfrac{\dots}{B}}{C} \to E}{\dfrac{\dfrac{C}{A \to C} \to I}{(A \to B) \to A \to C} \to I}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

then, for assumptions Γ*:*

$$\frac{\dfrac{\dfrac{[A \to B \to C] \quad [A]}{C} \to E \quad \dfrac{\dots}{B}}{C} \to E}{\dfrac{\dfrac{C}{A \to C} \to I}{(A \to B) \to A \to C} \to I}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

then, for assumptions Γ*:*

$$\frac{\dfrac{\dfrac{[A \to B \to C] \quad [A]}{C} \to E \quad \dfrac{[A \to B] \quad \dfrac{\dots}{A}}{B} \to E}{C} \to E}{\dfrac{\dfrac{C}{A \to C} \to I}{(A \to B) \to A \to C} \to I}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

and finally, still for assumptions Γ*:*

$$\frac{\dfrac{\dfrac{[A \to B \to C] \quad [A]}{C} \to E \quad \dfrac{[A \to B] \quad [A]}{B} \to E}{C} \to E}{\dfrac{\dfrac{C}{A \to C} \to I}{(A \to B) \to A \to C} \to I}{(A \to B \to C) \to (A \to B) \to A \to C} \to I$$

4. (System F) We have the Church numerals $\underline{n} = \lambda fx.f^n x$ in the untyped λ-calculus, e.g. $\underline{1} = \lambda fx.fx$ and $\underline{2} = \lambda fx.f(fx)$. Addition, multiplication, and exponentiation of two numerals can be defined by the respective λ-terms $\lambda nmfx.nf(mfx)$, $\lambda nmfx.nf(mf)x$, $\lambda nm.nm$.

   - Show that these definitions are ok, by testing them on the inputs $\underline{1}$ and $\underline{2}$.

   We do this only for the application to $\underline{2}$ and $\underline{1}$.

   $$(\lambda nmfx.nf(mfx))\underline{2}\underline{1} \twoheadrightarrow \lambda fx.\underline{2}f(\underline{1}fx) \twoheadrightarrow \lambda fx.f(f(\underline{1}fx)) \twoheadrightarrow \lambda fx.f(f(fx)) = \underline{3} = \underline{2+1}$$

   The λ-term for multiplication above contains a typo (easily found when doing the next item(s)), namely a spurious f. Correcting it, we have:

   $$(\lambda nmfx.n(mf)x)\underline{2}\underline{1} \twoheadrightarrow \lambda fx.\underline{2}(\underline{1}f)x \twoheadrightarrow \lambda fx.\underline{1}f(\underline{1}fx) \twoheadrightarrow \lambda fx.\underline{1}f(fx) \twoheadrightarrow \lambda fx.f(fx) = \underline{2} = \underline{2 \cdot 1}$$

   $$(\lambda nm.nm)\underline{2}\underline{1} \twoheadrightarrow \underline{2}\underline{1} \to \lambda x.\underline{1}(\underline{1}x) \to \lambda xy.\underline{1}xy \twoheadrightarrow \lambda xy.xy = \underline{1} = \underline{1^2}$$

   The principal type of Church numerals in the simply typed λ-calculus is $(A \to A) \to (A \to A)$.

   - Show that when restriction to a fixed type A in the above, addition and multiplication can be typed appropriately, but exponentiation cannot. Can you fix this (how?) when the restriction is dropped?

     Abbreviating $(A \to A) \to (A \to A)$ to N, one easily checks $\lambda n : N, m : N, f : A \to A, x : A.nf(mfx)$ has type $N \to N \to N$, since $mfx : A$ and hence $nf(mfx) : A$ as well.

     $\lambda n : N, m : N, f : A \to A, x : A.n(mf)x$ has type $N \to N \to N$, since $mf : A \to A$ and hence $n(mf)x : A$.

     $\lambda n : N, m : N.nm$ does not type-check, since nm doesn't.

     It can be fixed by setting $n : N \to N$, or, stated differently by instantiating A to $A \to A$ in the type schema, in the case of n.

   In the polymorphic λ-calculus ($\lambda 2$) the Church numerals take a type as first (extra) parameter and are of type $(\forall A)(A \to A) \to (A \to A)$. In Coq it and addition can be defined accordingly by

```
Definition cnat := forall X : Type, (X -> X) -> X -> X.
Definition addition  (n m: cnat) : cnat :=
fun (X : Type)(f : X -> X)(x : X) => n X f (m X f x).
```

   - Do the first item above in Coq, i.e. define multiplication and exponentiation and verify that they are well-defined, e.g. by proving that 2 two times 1 is 2.

     We continue the above two definitions by:

```
Definition one :cnat := fun (X:Type)(f : X -> X)(x : X) => f x.
Definition two :cnat := fun (X:Type)(f : X -> X)(x : X) => f (f x).
Definition three :cnat := fun (X:Type)(f : X -> X)(x : X) => f(f (f x)).
Lemma add21 : addition two one = three.
Proof. trivial. Qed.
Print add21.
Definition multiplication   (n m: cnat) : cnat :=
fun (X : Type)(f : X -> X)(x : X) => n X (m X f) x.
Lemma mul21 : multiplication two one = two.
Proof. trivial. Qed.
Print mul21.
Definition exponentiation   (n m: cnat) : cnat :=
fun (X : Type) => n (X->X) (m X) .
Lemma exp21 : exponentiation two one = one.
Proof. trivial. Qed.
Print exp21.
```

     Remark: the proofs are printed to illustrate that all results are proven by reflexivity; the computation (reduction) is done implicitly.

- Can you give the λ-terms in β-normal form inhabiting the type of Church numerals. Is each of them a natural number?

  All terms in β-normal form inhabiting the type of Church numerals have shape $\lambda f x.f^n x$, except for the term $\lambda f.f$ (the identity functional).

  Remark: this can be prevented by swapping the f and the x, i.e. having numerals of shape $\lambda x f.f^n x$ of type $(\forall A)A \to (A \to A) \to A$.

5. Prove in Coq the result on the correspondence between the square of sum and the sum of cubes (see the PS two weeks ago), again using elementary proof steps and lemmas. What are the differences between this proof and your earlier HOL Light proof, if any?

   A proof in basic style employing many auxiliary definitions and lemmas is:

```
Require Import Coq.Arith.Mult.
Definition double (n:nat) : nat := n + n.
Lemma doubleplus : forall n m:nat, double (n+m) = double n + double m.
Proof.
unfold double.
intros.
rewrite <- plus_assoc.
rewrite <- plus_assoc.
f_equal.
rewrite plus_comm.
rewrite <- plus_assoc.
f_equal.
Qed.
Lemma doublemult : forall n m, double (n * m) = double n * m.
Proof.
intros.
unfold double.
rewrite mult_plus_distr_r.
trivial.
Qed.
Fixpoint lines (n:nat) : nat :=
match n with
| 0 => 0
| S n => (lines n) + (S n)
end.
Lemma doublelines : forall n, double (lines n) = n * S n.
Proof.
induction n.
auto.
simpl.
rewrite doubleplus.
rewrite IHn.
unfold double.
simpl.
rewrite plus_comm.
simpl.
f_equal.
rewrite <- plus_assoc.
rewrite plus_comm.
simpl.
f_equal.
rewrite <- plus_assoc.
f_equal.
rewrite mult_comm.
replace (n * S (S n)) with (S (S n) * n).
simpl.
rewrite <- plus_assoc.
f_equal.
```

```
rewrite plus_comm.
trivial.
rewrite mult_comm.
trivial.
Qed.
Definition square (n: nat) : nat := n * n.
Lemma squareplus : forall n m, square (n + m) = square n + double (n * m) + square m.
Proof.
unfold square.
unfold double.
intros.
rewrite mult_plus_distr_l.
rewrite mult_plus_distr_r.
rewrite <- plus_assoc.
rewrite <- plus_assoc.
f_equal.
rewrite mult_plus_distr_r.
rewrite plus_comm.
rewrite <- plus_assoc.
rewrite <- plus_assoc.
f_equal.
rewrite plus_comm.
f_equal.
rewrite mult_comm.
trivial.
Qed.
Definition cube (n:nat) : nat := n * n * n.
Definition squares (n: nat) : nat := square (lines n).
Fixpoint cubes  (n:nat) : nat :=
match n with
| 0 => 0
| S n => plus (cubes n) (cube (S n))
end.
Lemma squarecube : forall n, squares n = cubes n.
Proof.
induction n.
auto.
unfold squares.
simpl.
rewrite squareplus.
fold (squares n).
rewrite <- plus_assoc.
f_equal.
assumption.
rewrite doublemult.
rewrite doublelines.
unfold square.
unfold cube.
simpl.
rewrite plus_comm.
simpl.
f_equal.
rewrite <- plus_assoc.
f_equal.
rewrite mult_plus_distr_r.
trivial.
Qed.
```

As shown in the PS a lot more automation is possible, e.g. using the ring tactic.