



1. Give an example of a proof that is incorrect because the side-condition for the introduction rule for \forall is violated.

2. The rule for elimination of an existential quantifier is:

$$\frac{\exists x. A \quad \forall x. (A \rightarrow B)}{B} E\exists$$

What is the side-condition for this rule?

3. Show that the following is a tautology of first-order intuitionistic predicate logic.

$$(\forall x. \neg P(x)) \rightarrow \neg(\exists x. P(x))$$

4. Show that the following is a tautology of first-order intuitionistic predicate logic.

$$\forall x. (P(x) \rightarrow \neg \forall y. (\neg P(y))).$$

5. Show that the following is a tautology of first-order intuitionistic predicate logic.

$$(\forall x. P(x)) \rightarrow \neg \exists y. \neg P(y).$$

6. Show that the following is a tautology of first-order intuitionistic predicate logic.

$$((\exists x. P(x)) \rightarrow (\forall y. Q(y))) \rightarrow \forall z. (P(z) \rightarrow Q(z)).$$

7. This exercise is concerned with λ -calculus with dependent types (λP).

A typing rule that is characteristic for λP is the following:

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi x:A. B : \square}$$

Explain how this rule is used to infer that the type of `natlist_dep` is ok.

8. This exercise is concerned with λ -calculus with dependent types (λP). Another typing rule that is characteristic for λP is the conversion rule:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{with } B =_{\beta} B'$$

Explain with an example (for instance `natlist_dep`) how the conversion rule can be used.

9. Give an inhabitant of $(\Pi x:\text{Terms}. P x) \rightarrow (P M)$.
10. Give an inhabitant of $(\Pi x:\text{Terms}. P x \rightarrow Q x) \rightarrow (\Pi x:\text{Terms}. P x) \rightarrow (\Pi y:\text{Terms}. Q x)$.
11. (This exercise is concerned with the Curry-Howard-De Bruijn isomorphism between first-order predicate logic and λP .)
Give the encoding of algebraic terms (from predicate logic) in λP .
12. (This exercise is concerned with the Curry-Howard-De Bruijn isomorphism between first-order predicate logic and λP .)
Give the encoding of formulas from predicate logic in λP .
13. (This exercise is concerned with the Curry-Howard-De Bruijn isomorphism between first-order predicate logic and λP .)
How are the introduction rules (for \rightarrow and \forall) from predicate logic represented in λP ?
14. (This exercise is concerned with the Curry-Howard-De Bruijn isomorphism between first-order predicate logic and λP .)
How are the elimination rules (for \rightarrow and \forall) from predicate logic represented in λP ?
15. First-order propositional logic can be encoded in Coq using dependent types as follows:
- ```
(* prop representing the propositions is a Set *)
Variable prop:Set.
(* implication on prop is a binary operator *)
Variable imp: prop -> prop -> prop.
```

```
(* T expresses if a proposition in prop is valid
 if (T p) is inhabited then p is valid
 if (T p) is not inhabited then p is not valid *)
Variable T: prop -> Prop.
```

Give the types of the variables `imp_introduction` and `imp_elimination` modelling the introduction- and elimination rule of implication.

16. This exercise is concerned with dependent types. We use the following definition in Coq:

```
Inductive natlist_dep : nat -> Set :=
 | nil_dep : natlist_dep 0
 | cons_dep : forall n : nat,
 nat -> natlist_dep n -> natlist_dep (S n).
```

- (a) What is the type of `natlist_dep`?  
 What is the type of `natlist_dep 2`?  
 Describe the elements of `natlist_dep 2`.
- (b) Suppose we want to define a function `nth` that takes as input a list and gives back the  $n$ th element of that list. How can dependent lists be used to avoid errors?
17. Give the type of `append_dep`, the function that appends two dependent lists. We use the following definition:

```
Fixpoint append_dep
 (n1:nat) (n2:nat)
 (l1:natlist_dep n1) (l2:natlist_dep n2)
 {struct l1} : natlist_dep (n1 + n2) :=
match l1 in (natlist_dep n1) return (natlist_dep (n1 + n2)) with
| nil_dep => l
| cons_dep p h t => cons_dep (p + n2) h (append_dep p t n2 l)
end.
```

18. Give the type of `reverse_dep`, the function that reverses a dependent list. We use the following definition:

```
Fixpoint reverse_dep
 (n:nat) (l:natlist_dep n) {struct l} :
 natlist_dep n :=
match l in (natlist_dep n) return (natlist_dep n) with
| nil_dep => nil_dep
| cons_dep p h t =>
 eq_rec (plus p 1) (fun n => natlist_dep n)
```

```
(append_dep p (reverse_dep p t) 1 (cons_dep 0 h nil_dep))
(S p) (P p)
end.
```

where

```
P : forall p : nat, p + 1 = S p
```

so P is a proof that  $p + 1$  equals  $S p$  for all  $p$ .

19. Consider the following two terms:

```
reverse_dep (plus n1 n2) (append_dep n1 n2 l1 l2)
append_dep n2 n1 (reverse_dep n2 l2) (reverse_dep n1 l1)
```

(Here  $n1$  and  $n2$  have type `nat`, the term `l1` has type `natlist_dep n1`, the term `l2` has type `natlist_dep n2`.)

What are the types of the above terms?

Are the types convertible?