



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

Answer: The following proof is *not* correct:

$$\frac{\frac{[P(x)^w]}{\forall x. P(x)} I\forall}{P(x) \rightarrow \forall x. P(x)} I[w] \rightarrow$$

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?

Answer:

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

is: x not free in B .

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

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

Answer: A proof of $(\forall x. \neg P(x)) \rightarrow \neg(\exists x. P(x))$:

$$\frac{\frac{\frac{[\exists x. P(x)^v]}{\perp} \quad [\forall x. P(x) \rightarrow \perp^u]}{\neg \exists x. P(x)} I[v] \rightarrow}{(\forall x. \neg P(x)) \rightarrow \neg(\exists x. P(x))} I[u] \rightarrow E\exists$$

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

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

Answer: A proof of $\forall x. (P(x) \rightarrow \neg \forall y. (\neg P(y)))$:

$$\begin{array}{c}
\frac{[(\forall y. (P(y) \rightarrow \perp))^v]}{P(x) \rightarrow \perp} E\forall \\
\frac{[P(x)^u]}{\frac{P(x) \rightarrow \perp}{\perp}} E \rightarrow \\
\frac{\perp}{(\forall y. (P(y) \rightarrow \perp)) \rightarrow \perp} I[v] \rightarrow \\
\frac{\neg \forall y. \neg P(y)}{P(x) \rightarrow \neg \forall y. (\neg P(y))} I[u] \rightarrow \\
\frac{P(x) \rightarrow \neg \forall y. (\neg P(y))}{\forall x. (P(x) \rightarrow \neg \forall y. (\neg P(y)))} I\forall
\end{array}$$

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

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

Answer:

A proof of $(\forall x. P(x)) \rightarrow \neg \exists y. \neg P(y)$:

$$\begin{array}{c}
\frac{[(\forall x. P(x))^u]}{P(y)} E \rightarrow \\
\frac{[(P(y) \rightarrow \perp)^w]}{\frac{P(y)}{\perp}} I[w] \\
\frac{\perp}{(P(y) \rightarrow \perp) \rightarrow \perp} I\forall \\
\frac{[(\exists y (P(y) \rightarrow \perp))^v]}{\frac{(P(y) \rightarrow \perp) \rightarrow \perp}{\forall y. (P(y) \rightarrow \perp) \rightarrow \perp}} E\exists \\
\frac{\perp}{(\exists y. (P(y) \rightarrow \perp)) \rightarrow \perp} I[v] \\
\frac{\neg \exists y. \neg P(y)}{(\forall x. P(x)) \rightarrow \neg \exists y. \neg P(y)} I[u] \rightarrow
\end{array}$$

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

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

Answer:

A proof of $((\exists x. P(x)) \rightarrow (\forall y. Q(y))) \rightarrow \forall z. (P(z) \rightarrow Q(z))$:

$$\begin{array}{c}
\frac{[(\exists x. P(x)) \rightarrow (\forall y. Q(y))]^a}{\frac{[P(z)^b]}{\exists x. P(x)}} E \rightarrow \\
\frac{\forall y. Q(y)}{Q(z)} E\forall \\
\frac{Q(z)}{P(z) \rightarrow Q(z)} I[b] \\
\frac{P(z) \rightarrow Q(z)}{\forall z. (P(z) \rightarrow Q(z))} I\forall \\
\frac{\forall z. (P(z) \rightarrow Q(z))}{((\exists x. P(x)) \rightarrow (\forall y. Q(y))) \rightarrow \forall z. (P(z) \rightarrow Q(z))} I[a]
\end{array}$$

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.

Answer:

The type of `natlist_dep` is `nat -> Set`. In λP notation: `nat` \rightarrow `*`, or, equivalently, $\Pi x : \text{nat}. *$. We use the rule with `nat` for A and `*` for B to infer that $\Pi x : \text{nat}. * : \square$:

$$\frac{\Gamma \vdash \text{nat} : * \quad \Gamma, x : \text{nat} \vdash * : \square}{\Gamma \vdash \Pi x : \text{nat}. * : \square}$$

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.

Answer:

Suppose that we have $\Gamma \vdash \text{nil} : \text{natlistdep } 0$. The conversion rule can then for instance be used to infer that $\Gamma \vdash \text{nil} : \text{natlistdep } ((\lambda x : \text{nat}. x) 0)$, taking `nil` for A and `natlistdep 0` for B and `natlistdep (($\lambda x : \text{nat}. x$) 0)` for B' :

$$\frac{\Gamma \vdash \text{nil} : \text{natlistdep } 0 \quad \Gamma \vdash \text{natlistdep } ((\lambda x : \text{nat}. x) 0) : *}{\Gamma \vdash \text{nil} : \text{natlistdep } ((\lambda x : \text{nat}. x) 0)}$$

9. Give an inhabitant of $(\Pi x : \text{Terms}. P x) \rightarrow (P M)$.

Answer: An inhabitant of $(\Pi x : \text{Terms}. P x) \rightarrow (P M)$:

$$\lambda u : (\Pi x : \text{Terms}. P x). (u M)$$

10. Give an inhabitant of $(\Pi x : \text{Terms}. P x \rightarrow Q x) \rightarrow (\Pi x : \text{Terms}. P x) \rightarrow (\Pi y : \text{Terms}. Q x)$.

Answer:

An inhabitant of $(\Pi x : \text{Terms}. P x \rightarrow Q x) \rightarrow (\Pi x : \text{Terms}. P x) \rightarrow (\Pi y : \text{Terms}. Q x)$:

$$\lambda u : (\Pi x : \text{Terms}. P x \rightarrow Q x). \lambda v : (\Pi x : \text{Terms}. P x). \lambda y : \text{Terms}. (u y (v y))$$

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 .

Answer: We have $\text{Terms} : \star$. Then, for every n -ary function symbol f in \mathcal{F} there is a distinguished variable f of type $\text{Terms} \rightarrow \dots \rightarrow \text{Terms} \rightarrow \text{Terms}$ with $n + 1$ times Terms .

An algebraic term is encoded in λP as follows:

- A variable x is encoded as $x : \text{Terms}$.
- A term $f(t_1, \dots, t_n)$ is encoded as $f t_1^* \dots t_n^*$ with t_i^* the encoding of t_i .

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 .

Answer: Formulas are translated into λP as follows:

- The atomic formula $R(M_1, \dots, M_n)$ in predicate logic is translated into the λ -term $R M_1^* \dots M_n^*$ with M_1^*, \dots, M_n^* the translations of M_1, \dots, M_n . Note that $R : \text{Terms} \rightarrow \dots \rightarrow \text{Terms} \rightarrow \star$ with n times Terms .
- The formula $A \rightarrow B$ in predicate logic is translated into the λ -term $\Pi x:A^*. B^*$, also written as $A^* \rightarrow B^*$.
- The formula $\forall x.A$ in predicate logic is translated into the λ -term $\Pi x:\text{Terms}. A^*$ with A^* the translation of A .

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 ?

Answer: The introduction rules in (minimal) predicate logic correspond to the abstraction rule 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 ?

Answer: The elimination rules in (minimal) predicate logic correspond to the application rule 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.

Answer:

```
imp_introduction : forall p q : prop, (T p -> T q) -> T (p => q).
```

```
imp_elimination : forall p q : prop, T (p => q) -> T p -> T q.
```

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`.

Answer:

The type of `natlist_dep` is `nat -> Set`.

The type of `natlist_dep 2` is `Set`.

The elements of `natlist_dep 2` are the lists consisting of two natural numbers.

- (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?

Answer:

Intuitively: the type indicates whether the list is long enough to take the `n`th element.

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.

```

Answer:

Then the type of `append_dep` is:

```
forall n1 n2 : nat,  
natlist_dep n1 -> natlist_dep n2 -> natlist_dep (n1 + n2)
```

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`.

Answer:

Then the type of `reverse_dep` is:

```
forall n : nat, natlist_dep n -> natlist_dep n
```

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?

Answer:

We have:

```
reverse_dep (n1 + n2) (append_dep n1 n2 l1 l2)  
  : natlist_dep (n1 + n2)
```

and

```
append_dep n2 n1 (reverse_dep n2 l2) (reverse_dep n1 l1)
  : natlist_dep (n2 + n1)
```

Those types are not convertible.