Typing in Coq

Daria Walukiewicz-Chrząszcz

09 April 2024

Proving in Coq

Curry-Howard isomorphism

proof of a given formula

⇕ ⇕

term of the corresponding type

$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ :\ (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

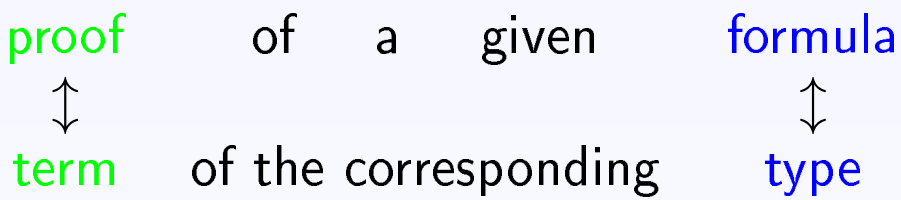
proof	of	a	given	formula
↕				↕
term	of the corresponding			type

$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ :\ (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism



$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ : \ (A \to B \to C) \to (A \to B) \to (A \to C)$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof of a given formula
↕ ↕
term of the corresponding type

$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ : \ (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4+2+4 typing rules for inductive types

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4+2+4 typing rules for inductive types

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4+2+4 typing rules for inductive types

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4+2+4 typing rules for inductive types

Coq — sorts

- Sorts in Coq:

$$\begin{array}{l} \mathit{Set} \\ \mathit{Prop} \quad : \mathit{Type}(1) : \mathit{Type}(2) : \ldots \\ \mathit{SProp} \end{array}$$

- Cummulativity (or sub-sorting):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}(1) \leq \mathit{Type}(2) \leq \ldots$$

Coq — sorts

- Sorts in Coq:

$$\begin{array}{ll} \mathit{Set} & \\ \mathit{Prop} & : \mathit{Type}(1) : \mathit{Type}(2) : \ldots \\ \mathit{SProp} & \end{array}$$

- Cummulativity (or sub-sorting):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}(1) \leq \mathit{Type}(2) \leq \ldots$$

Coq — sorts Prop and Set

$$d : A : Prop$$

A is a formula, d is a proof of A

$$n : T : Set$$

T is a type, n is a value of type T

Coq — sorts Prop and Set

$$d : A : Prop$$

A is a formula, d is a proof of A

$$n : T : Set$$

T is a type, n is a value of type T

Coq — sorts Prop and Set

$$d : A : Prop$$

A is a formula, d is a proof of A

$$n : T : Set$$

T is a type, n is a value of type T

Coq — sorts Prop and Set

$$d : A : Prop$$

`A` is a formula, `d` is a proof of `A`

$$n : T : Set$$

`T` is a type, `n` is a value of type `T`

Coq — abstraction and application

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ to $\forall x{:}A.B$, where $x \notin FV(B)$

application rule:

$$\frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — abstraction and application

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ to $\forall x{:}A.B$, where $x \notin FV(B)$

application rule:

$$\frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — abstraction and application

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ to $\forall x{:}A.B$, where $x \notin FV(B)$

application rule:

$$\frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — products

$$\frac{\Gamma \vdash A : s \qquad s \texttt{ is a sort} \qquad \Gamma, x{:}A \vdash B : Prop}{\Gamma \vdash \forall x{:}A.B : Prop}$$ **Prod-Prop**

$$\frac{\Gamma \vdash A : s \qquad s \texttt{ is a sort} \qquad \Gamma, x{:}A \vdash B : SProp}{\Gamma \vdash \forall x{:}A.B : SProp}$$ **Prod-SProp**

$$\frac{\Gamma \vdash A : s \qquad s \in \{Set, Prop, SProp\} \qquad \Gamma, x{:}A \vdash B : Set}{\Gamma \vdash \forall x{:}A.B : Set}$$ **Prod-Set**

$$\frac{\Gamma \vdash A : s \quad s \in \{Type(i), SProp\} \qquad \Gamma, x{:}A \vdash B : Type(i)}{\Gamma \vdash \forall x{:}A.B : Type(i)}$$ **Prod-Type**

`Prop` is impredicative, `Set`, `Type` are predicative

Coq — products

$$\frac{\Gamma \vdash A : s \qquad s \texttt{ is a sort} \qquad \Gamma, x{:}A \vdash B : Prop}{\Gamma \vdash \forall x{:}A.B : Prop}$$ **Prod-Prop**

$$\frac{\Gamma \vdash A : s \qquad s \texttt{ is a sort} \qquad \Gamma, x{:}A \vdash B : SProp}{\Gamma \vdash \forall x{:}A.B : SProp}$$ **Prod-SProp**

$$\frac{\Gamma \vdash A : s \qquad s \in \{Set, Prop, SProp\} \qquad \Gamma, x{:}A \vdash B : Set}{\Gamma \vdash \forall x{:}A.B : Set}$$ **Prod-Set**

$$\frac{\Gamma \vdash A : s \quad s \in \{Type(i), SProp\} \qquad \Gamma, x{:}A \vdash B : Type(i)}{\Gamma \vdash \forall x{:}A.B : Type(i)}$$ **Prod-Type**

`Prop` is impredicative, `Set`, `Type` are predicative

Coq — examples of product types

- functional type $\texttt{nat} \to \texttt{nat}$

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{nat} \ : \ \texttt{Set}}{\Gamma \vdash \texttt{nat} \to \texttt{nat} \ : \ \texttt{Set}}\ \textbf{Prod-Set}$$

- type of a type constructor (ex: `List`)

$$\frac{\Gamma \vdash \texttt{Set} \ : \ \texttt{Type} \qquad \Gamma, x : \texttt{Set} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{Set} \to \texttt{Set} \ : \ \texttt{Type}}\ \textbf{Prod-Type}$$

- type of a predicate (ex: `Even`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Prop} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Prop} \ : \ \texttt{Type}}\ \textbf{Prod-Type}$$

Coq — examples of product types

- functional type $\texttt{nat} \to \texttt{nat}$

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{nat} \ : \ \texttt{Set}}{\Gamma \vdash \texttt{nat} \to \texttt{nat} \ : \ \texttt{Set}}\textbf{Prod-Set}$$

- type of a type constructor (ex: `List`)

$$\frac{\Gamma \vdash \texttt{Set} \ : \ \texttt{Type} \qquad \Gamma, x : \texttt{Set} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{Set} \to \texttt{Set} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

- type of a predicate (ex: `Even`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Prop} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Prop} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

Coq — examples of product types

- functional type $\texttt{nat} \to \texttt{nat}$

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{nat} \ : \ \texttt{Set}}{\Gamma \vdash \texttt{nat} \to \texttt{nat} \ : \ \texttt{Set}}\textbf{Prod-Set}$$

- type of a type constructor (ex: `List`)

$$\frac{\Gamma \vdash \texttt{Set} \ : \ \texttt{Type} \qquad \Gamma, x : \texttt{Set} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{Set} \to \texttt{Set} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

- type of a predicate (ex: `Even`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Prop} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Prop} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

Coq — examples of product types cont.

- type of a dependent type (ex: `ftree`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Set} \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- type of "polymorphic" identity: $\forall \alpha : \texttt{Set}. \alpha \to \alpha \ : \ \texttt{Type}$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Type}}{\Gamma \vdash \forall \alpha : \texttt{Set}. \alpha \to \alpha \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \texttt{Prop} \ : \ \texttt{Type} \qquad \Gamma, \alpha : \texttt{Prop} \vdash \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}{\Gamma \vdash \forall \alpha : \texttt{Prop}. \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}$$

Coq — examples of product types cont.

- type of a dependent type (ex: `ftree`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Set} \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- type of "polymorphic" identity: $\forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Type}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \texttt{Prop} \ : \ \texttt{Type} \qquad \Gamma, \alpha : \texttt{Prop} \vdash \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}{\Gamma \vdash \forall \alpha : \texttt{Prop}.\alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}$$

Coq — examples of product types cont.

- type of a dependent type (ex: `ftree`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Set} \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- type of "polymorphic" identity: $\forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Type}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \texttt{Prop} \ : \ \texttt{Type} \qquad \Gamma, \alpha : \texttt{Prop} \vdash \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}{\Gamma \vdash \forall \alpha : \texttt{Prop}.\alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}$$

Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; simpl; reflexivity.
Qed.
```

Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; simpl; reflexivity.
Qed.
```

Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; simpl; reflexivity.
Qed.
```

Natural numbers — induction

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal:

plus O O = O

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

Theorem n_plus_O : $\forall$ n : **nat**, plus n O = n.
  induction n.

The first subgoal:

plus O O = O

  reflexivity.

holds because of conversion (iota reduction). The second is:

  n : **nat**
  IHn : plus n O = n
  ============================
   plus (S n) O = S n

simpl.
rewrite *IHn*.
reflexivity.
Qed.

Natural numbers — induction

Theorem n_plus_O : ∀ *n* : **nat**, plus *n* O = *n*.

```
  induction n.
```

The first subgoal:

plus O O = O

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

n : **nat**
IHn : plus *n* O = *n*
==========
plus (S *n*) O = S *n*

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal:

plus O O = O

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n

simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

Theorem n_plus_O : ∀ *n* : **nat**, plus *n* O = *n*.
  induction *n*.

The first subgoal:

plus O O = O

  reflexivity.

holds because of conversion (iota reduction). The second is:

  n : **nat**
  IHn : plus *n* O = *n*
  ============================
   plus (S *n*) O = S *n*

simpl.
rewrite *IHn*.
reflexivity.
Qed.

Natural numbers — induction principle

```
Check nat_ind.
```

```
nat_ind : ∀ P : nat → Prop,
            P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

```
Theorem n_plus_O' : ∀ n : nat, plus n O = n.
  apply (nat_ind (fun n ⇒ plus n O = n));
[reflexivity | intros n IHn; simpl; rewrite IHn; reflexivity].
Qed.
```

Natural numbers — induction principle

```
Check nat_ind.
```

```
  nat_ind : ∀ P : nat → Prop,
              P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

```
Theorem n_plus_O' : ∀ n : nat, plus n O = n.
  apply (nat_ind (fun n ⇒ plus n O = n));
[reflexivity | intros n IHn; simpl; rewrite IHn; reflexivity].
Qed.
```

Natural numbers — induction principle and recursors

```
Print nat_ind.

nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

Print nat_rec.

nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

Check nat_rect.

nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Natural numbers — induction principle and recursors

```
Print nat_ind.

nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

Print nat_rec.

nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

Check nat_rect.

nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Natural numbers — induction principle and recursors

```
Print nat_ind.

nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

Print nat_rec.

nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

Check nat_rect.

nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Primitives fix and match

```
Print nat_rect.

nat_rect =
fun (P : nat → Type) (f : P O) (f0 : ∀ n : nat, P n → P (S n)) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | O ⇒ f
  | S n' ⇒ f0 n' (F n')
  end
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Parametric lists

```
Inductive list (T : Set) : Set :=
| Nil : list T
| Cons : T → list T → list T.
```

```
Check list_ind.
```

```
list_ind
     : ∀ (T : Set) (P : list T → Prop),
       P (Nil T) →
       (∀ (t : T) (l : list T), P l → P (Cons T t l)) →
       ∀ l : list T, P l
```

Parametric lists

```
Inductive list (T : Set) : Set :=
| Nil : list T
| Cons : T → list T → list T.
```

```
Check list_ind.
```

```
list_ind
     : ∀ (T : Set) (P : list T → Prop),
       P (Nil T) →
       (∀ (t : T) (l : list T), P l → P (Cons T t l)) →
       ∀ l : list T, P l
```

Parametric lists cont.

```
Arguments Nil {T}.
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons x ls1' ⇒ Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Parametric lists cont.

```
Arguments Nil {T}.
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons x ls1' ⇒ Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Parametric lists cont.

```
Arguments Nil {T}.
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons x ls1' ⇒ Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Parametric lists cont.

```
Arguments Nil {T}.
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons x ls1' ⇒ Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Nonparametric lists

```
Inductive lista : Set -> Type :=
| Nila : forall (A:Set), lista A
| Consa : forall (A:Set), A -> lista A -> lista A.
```

```
Check lista_ind.
```

```
lista_ind:
    forall P : (forall A : Set, lista A -> Prop),
    (forall A : Set, P A (Nila A)) ->
    (forall (A : Set) (a : A) (l : lista A),
          P A l -> P A (Consa A a l)) ->
    forall (P0 : Set) (l : lista P0), P P0 l
```

Nonparametric lists

```
Inductive lista : Set -> Type :=
| Nila : forall (A:Set), lista A
| Consa : forall (A:Set), A -> lista A -> lista A.
```

```
Check lista_ind.
```

```
lista_ind:
    forall P : (forall A : Set, lista A -> Prop),
    (forall A : Set, P A (Nila A)) ->
    (forall (A : Set) (a : A) (l : lista A),
          P A l -> P A (Consa A a l)) ->
    forall (P0 : Set) (l : lista P0), P P0 l
```

Nonparametric lists

```
Inductive lista : Set -> Type :=
| Nila : forall (A:Set), lista A
| Consa : forall (A:Set), A -> lista A -> lista A.


Check lista_ind.


lista_ind:
    forall P : (forall A : Set, lista A -> Prop),
    (forall A : Set, P A (Nila A)) ->
    (forall (A : Set) (a : A) (l : lista A),
          P A l -> P A (Consa A a l)) ->
    forall (P0 : Set) (l : lista P0), P P0 l
```

Trees

```
Inductive nat_btree : Set :=
| NLeaf : nat_btree
| NNode : nat_btree → nat → nat_btree → nat_btree.
```

```
Check nat_btree_ind.

nat_btree_ind
     : ∀ P : nat_btree → Prop,
       P NLeaf →
       (∀ n : nat_btree, P n → ∀ (n0 : nat) (n1 : nat_btree),
        P n1 → P (NNode n n0 n1)) →
       ∀ n : nat_btree, P n
```

Trees

```
Inductive nat_btree : Set :=
| NLeaf : nat_btree
| NNode : nat_btree → nat → nat_btree → nat_btree.
```

```
Check nat_btree_ind.
```

```
nat_btree_ind
     : ∀ P : nat_btree → Prop,
       P NLeaf →
       (∀ n : nat_btree, P n → ∀ (n0 : nat) (n1 : nat_btree),
        P n1 → P (NNode n n0 n1)) →
       ∀ n : nat_btree, P n
```

Mutually recursive types: odd_list and even_list

```
Inductive even_list : Set :=
| ENil : even_list
| ECons : nat → odd_list → even_list

with odd_list : Set :=
| OCons : nat → even_list → odd_list.
```

```
Check even_list_ind.

  even_list_ind
     : ∀ P : even_list → Prop,
       P ENil →
       (∀ (n : nat) (o : odd_list), P (ECons n o)) →
       ∀ e : even_list, P e
```

Mutually recursive types: odd_list and even_list

```
Inductive even_list : Set :=
| ENil : even_list
| ECons : nat → odd_list → even_list

with odd_list : Set :=
| OCons : nat → even_list → odd_list.
```

```
Check even_list_ind.
```

```
even_list_ind
     : ∀ P : even_list → Prop,
       P ENil →
       (∀ (n : nat) (o : odd_list), P (ECons n o)) →
       ∀ e : even_list, P e
```

Scheme — generation of induction principles

```
Scheme even_list_mut := Induction for even_list Sort Prop
with odd_list_mut := Induction for odd_list Sort Prop.
```

```
Check even_list_mut.
```

```
even_list_mut
     : ∀ (P : even_list → Prop) (P0 : odd_list → Prop),
       P ENil →
       (∀ (n : nat) (o : odd_list), P0 o → P (ECons n o)) →
       (∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) →
       ∀ e : even_list, P e
```

Scheme — generation of induction principles

```
Scheme even_list_mut := Induction for even_list Sort Prop
with odd_list_mut := Induction for odd_list Sort Prop.
```

```
Check even_list_mut.
```

```
even_list_mut
     : ∀ (P : even_list → Prop) (P0 : odd_list → Prop),
       P ENil →
       (∀ (n : nat) (o : odd_list), P0 o → P (ECons n o)) →
       (∀ (n : nat) (e : even_list), P e → P0 (OCons n e)) →
       ∀ e : even_list, P e
```

Reflexive type: formula

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Forall : (nat → formula) → formula.
```

```
Check formula_ind.
```

```
formula_ind
     : ∀ P : formula → Prop,
       (∀ n n0 : nat, P (Eq n n0)) →
       (∀ f0 : formula,
        P f0 → ∀ f1 : formula, P f1 → P (And f0 f1)) →
       (∀ f1 : nat → formula,
        (∀ n : nat, P (f1 n)) → P (Forall f1)) →
       ∀ f2 : formula, P f2
```

Reflexive type: formula

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Forall : (nat → formula) → formula.
```

```
Check formula_ind.
```

```
formula_ind
     : ∀ P : formula → Prop,
       (∀ n n0 : nat, P (Eq n n0)) →
       (∀ f0 : formula,
        P f0 → ∀ f1 : formula, P f1 → P (And f0 f1)) →
       (∀ f1 : nat → formula,
        (∀ n : nat, P (f1 n)) → P (Forall f1)) →
       ∀ f2 : formula, P f2
```

Restrictions: positivity condition

```
Inductive term : Set :=
| App : term → term → term
| Abs : (term → term) → term.
```

```
Error: Non strictly positive occurrence of "term" in "(term
-> term) -> term"
```

Restrictions: positivity condition

```
Inductive term : Set :=
| App : term → term → term
| Abs : (term → term) → term.
```

```
Error: Non strictly positive occurrence of "term" in "(term
-> term) -> term"
```

Restrictions: only small inductive types in Set

Correct (definition of $\exists\, \phi\ \ P(\phi)$):

```
Inductive exProp (P:Prop->Prop) : Prop
:= exP_intro : forall X:Prop, P X -> exProp P.
```

Incorrect:

```
Inductive exSet (P:Set->Prop) : Set
:= exS_intro : forall X:Set, P X -> exSet P.
```

```
Error: Large non-propositional inductive types must be in Type.
```

Correct:

```
Inductive exType (P:Type->Prop) : Type
:= exT_intro : forall X:Type, P X -> exType P.
```

behind the scene: $k < j$ and $k \leq i$ (universe constraints)

$$\texttt{exType} : (P : \texttt{Type}_i \to \texttt{Prop}) \to \texttt{Type}_j$$

$$\texttt{exT\_intro} : \forall\, X : \texttt{Type}_k,\ P\, X \to \texttt{exType}\, P$$

Restrictions: only small inductive types in Set

Correct (definition of $\exists\, \phi\ \ P(\phi)$):

```
Inductive exProp (P:Prop->Prop) : Prop
:= exP_intro : forall X:Prop, P X -> exProp P.
```

Incorrect:

```
Inductive exSet (P:Set->Prop) : Set
:= exS_intro : forall X:Set, P X -> exSet P.
```

```
Error: Large non-propositional inductive types must be in Type.
```

Correct:

```
Inductive exType (P:Type->Prop) : Type
:= exT_intro : forall X:Type, P X -> exType P.
```

behind the scene: $k < j$ and $k \leq i$ (universe constraints)

$$\mathtt{exType} : (P : \mathtt{Type}_i \to \mathtt{Prop}) \to \mathtt{Type}_j$$

$$\mathtt{exT\_intro} : \forall\, X : \mathtt{Type}_k,\ P\, X \to \mathtt{exType}\, P$$

Restrictions: only small inductive types in Set

Correct (definition of $\exists\ \phi\ \ P(\phi)$):

```
Inductive exProp (P:Prop->Prop) : Prop
:= exP_intro : forall X:Prop, P X -> exProp P.
```

Incorrect:

```
Inductive exSet (P:Set->Prop) : Set
:= exS_intro : forall X:Set, P X -> exSet P.
```

```
Error: Large non-propositional inductive types must be in Type.
```

Correct:

```
Inductive exType (P:Type->Prop) : Type
:= exT_intro : forall X:Type, P X -> exType P.
```

behind the scene: $k < j$ and $k \leq i$ (universe constraints)

$$\texttt{exType} : (P : \texttt{Type}_i \to \texttt{Prop}) \to \texttt{Type}_j$$

$$\texttt{exT\_intro} : \forall\ X : \texttt{Type}_k,\ P\,X \to \texttt{exType}\,P$$

Definitional equality

```
Print "=".
```

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

```
Check @eq_refl.
@eq_refl
     : forall (A : Type) (x : A), x = x
```

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                          P x -> forall y : A, x = y -> P y
```

Definitional equality

```
Print "=".
```

Inductive **eq** $(A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq\_refl} : x = x$

```
Check @eq_refl.
@eq_refl
     : forall (A : Type) (x : A), x = x
```

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                          P x -> forall y : A, x = y -> P y
```

Definitional equality

```
Print "=".
```

Inductive **eq** $(A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq\_refl} : x = x$

```
Check @eq_refl.
@eq_refl
     : forall (A : Type) (x : A), x = x
```

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                          P x -> forall y : A, x = y -> P y
```

Definitional equality

```
Print "=".
```

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : x = x
```

```
Check @eq_refl.
@eq_refl
     : forall (A : Type) (x : A), x = x
```

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                          P x -> forall y : A, x = y -> P y
```

reflexivity and rewrite

```
reflexivity  ≡  apply eq_refl
```

```
rewrite H  ≡  apply eq_ind
  (where H: a=b)
```

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                        P x -> forall y : A, x = y -> P y
```

reflexivity and rewrite

```
reflexivity  ≡  apply eq_refl
```

```
rewrite H  ≡  apply eq_ind
```

(where H: a=b)

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
                          P x -> forall y : A, x = y -> P y
```