

Typing in Coq

Daria Walukiewicz-Chrząszcz

28 March 2023

Proving in Coq

Curry-Howard isomorphism

proof of a given formula
 \updownarrow
 term of the corresponding type

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

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof of a given formula
 \updownarrow
 term of the corresponding type

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

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof of a given **formula**
 \updownarrow
term of the corresponding **type**

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

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof of a given formula
 \updownarrow
 term of the corresponding type

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

Correctness of Coq relies on correctness of type-checking

Coq — typing judgment

`environment ⊢ term : 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

`environment ⊢ term : 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

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

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

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

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:

Set

Prop : *Type(1)*: *Type(2)*: ...

SProp

- Cummulativity (or sub-sorting):

Prop ≤ *Set* ≤ *Type(1)* ≤ *Type(2)* ≤ ...

Coq — sorts

- Sorts in Coq:

Set

Prop : *Type(1)*: *Type(2)*: ...

SProp

- Cummulativity (or sub-sorting):

$Prop \leq Set \leq Type(1) \leq Type(2) \leq \dots$

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: $A \rightarrow B$ to $\forall x:A.B$, where $x \notin FV(B)$

application rule:

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

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: $A \rightarrow B$ to $\forall x:A.B$, where $x \notin FV(B)$

application rule:

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

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: $A \rightarrow B$ 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

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

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

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

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

Prop is impredicative, Set, Type are predicative

Coq — products

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

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

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

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

Prop is impredicative, Set, Type are predicative

Coq — examples of product types

- functional type $\text{nat} \rightarrow \text{nat}$

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{nat} : \text{Set}}{\Gamma \vdash \text{nat} \rightarrow \text{nat} : \text{Set}} \text{Prod-Set}$$

- type of a type constructor (ex: List)

$$\frac{\Gamma \vdash \text{Set} : \text{Type} \quad \Gamma, x : \text{Set} \vdash \text{Set} : \text{Type}}{\Gamma \vdash \text{Set} \rightarrow \text{Set} : \text{Type}} \text{Prod-Type}$$

- type of a predicate (ex: Even)

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{Prop} : \text{Type}}{\Gamma \vdash \text{nat} \rightarrow \text{Prop} : \text{Type}} \text{Prod-Type}$$

Coq — examples of product types

- functional type $\text{nat} \rightarrow \text{nat}$

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{nat} : \text{Set}}{\Gamma \vdash \text{nat} \rightarrow \text{nat} : \text{Set}} \text{Prod-Set}$$

- type of a type constructor (ex: List)

$$\frac{\Gamma \vdash \text{Set} : \text{Type} \quad \Gamma, x : \text{Set} \vdash \text{Set} : \text{Type}}{\Gamma \vdash \text{Set} \rightarrow \text{Set} : \text{Type}} \text{Prod-Type}$$

- type of a predicate (ex: Even)

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{Prop} : \text{Type}}{\Gamma \vdash \text{nat} \rightarrow \text{Prop} : \text{Type}} \text{Prod-Type}$$

Coq — examples of product types

- functional type $\text{nat} \rightarrow \text{nat}$

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{nat} : \text{Set}}{\Gamma \vdash \text{nat} \rightarrow \text{nat} : \text{Set}} \text{Prod-Set}$$

- type of a type constructor (ex: List)

$$\frac{\Gamma \vdash \text{Set} : \text{Type} \quad \Gamma, x : \text{Set} \vdash \text{Set} : \text{Type}}{\Gamma \vdash \text{Set} \rightarrow \text{Set} : \text{Type}} \text{Prod-Type}$$

- type of a predicate (ex: Even)

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{Prop} : \text{Type}}{\Gamma \vdash \text{nat} \rightarrow \text{Prop} : \text{Type}} \text{Prod-Type}$$

Coq — examples of product types cont.

- type of a dependent type (ex: ftree)

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{Set} : \text{Type}}{\Gamma \vdash \text{nat} \rightarrow \text{Set} : \text{Type}} \text{Prod-Type}$$

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

$$\frac{\Gamma \vdash \text{Set} : \text{Type} \quad \Gamma, \alpha : \text{Set} \vdash \alpha \rightarrow \alpha : \text{Type}}{\Gamma \vdash \forall \alpha : \text{Set}. \alpha \rightarrow \alpha : \text{Type}} \text{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \text{Prop} : \text{Type} \quad \Gamma, \alpha : \text{Prop} \vdash \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : \text{Prop}}{\Gamma \vdash \forall \alpha : \text{Prop}. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : \text{Prop}}$$

Coq — examples of product types cont.

- type of a dependent type (ex: ftree)

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{Set} : \text{Type}}{\Gamma \vdash \text{nat} \rightarrow \text{Set} : \text{Type}} \text{Prod-Type}$$

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

$$\frac{\Gamma \vdash \text{Set} : \text{Type} \quad \Gamma, \alpha : \text{Set} \vdash \alpha \rightarrow \alpha : \text{Type}}{\Gamma \vdash \forall \alpha : \text{Set}. \alpha \rightarrow \alpha : \text{Type}} \text{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \text{Prop} : \text{Type} \quad \Gamma, \alpha : \text{Prop} \vdash \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : \text{Prop}}{\Gamma \vdash \forall \alpha : \text{Prop}. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : \text{Prop}}$$

Coq — examples of product types cont.

- type of a dependent type (ex: ftree)

$$\frac{\Gamma \vdash \text{nat} : \text{Set} \quad \Gamma, x : \text{nat} \vdash \text{Set} : \text{Type}}{\Gamma \vdash \text{nat} \rightarrow \text{Set} : \text{Type}} \text{Prod-Type}$$

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

$$\frac{\Gamma \vdash \text{Set} : \text{Type} \quad \Gamma, \alpha : \text{Set} \vdash \alpha \rightarrow \alpha : \text{Type}}{\Gamma \vdash \forall \alpha : \text{Set}. \alpha \rightarrow \alpha : \text{Type}} \text{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \text{Prop} : \text{Type} \quad \Gamma, \alpha : \text{Prop} \vdash \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : \text{Prop}}{\Gamma \vdash \forall \alpha : \text{Prop}. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : \text{Prop}}$$

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_0` : $\forall n : \mathbf{nat}$, `plus n 0 = n`.
`induction n`.

The first subgoal:

`plus 0 0 = 0`

`reflexivity`.

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

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl`.

`rewrite IHn`.

`reflexivity`.

`Qed`.

Natural numbers — induction

Theorem `n_plus_0` : $\forall n : \mathbf{nat}$, `plus n 0 = n`.
`induction n`.

The first subgoal:

`plus 0 0 = 0`

`reflexivity`.

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

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl`.

`rewrite IHn`.

`reflexivity`.

`Qed`.

Natural numbers — induction

Theorem `n_plus_0` : $\forall n : \mathbf{nat}$, `plus n 0 = n`.
`induction n`.

The first subgoal:

`plus 0 0 = 0`

`reflexivity`.

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

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl`.

`rewrite IHn`.

`reflexivity`.

`Qed`.

Natural numbers — induction

Theorem `n_plus_0` : $\forall n : \mathbf{nat}$, `plus n 0 = n`.
`induction n`.

The first subgoal:

`plus 0 0 = 0`

`reflexivity`.

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

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl`.

`rewrite IHn`.

`reflexivity`.

`Qed`.

Natural numbers — induction

Theorem `n_plus_0` : $\forall n : \mathbf{nat}$, `plus n 0 = n`.
`induction n`.

The first subgoal:

`plus 0 0 = 0`

`reflexivity`.

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

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl`.

`rewrite IHn`.

`reflexivity`.

`Qed`.

Natural numbers — induction principle

Check `nat_ind`.

$$\text{nat_ind} : \forall P : \mathbf{nat} \rightarrow \text{Prop},$$

$$P \ 0 \rightarrow (\forall n : \mathbf{nat}, P \ n \rightarrow P \ (S \ n)) \rightarrow \forall n : \mathbf{nat}, P \ n$$

Theorem `n_plus_0'` : $\forall n : \mathbf{nat}, \text{plus } n \ 0 = n$.

```

  apply (nat_ind (fun n => plus n 0 = n));
[reflexivity | intros n IHn; simpl; rewrite IHn; reflexivity].
Qed.
```

Natural numbers — induction principle

Check `nat_ind`.

$$\text{nat_ind} : \forall P : \mathbf{nat} \rightarrow \text{Prop},$$

$$P \ 0 \rightarrow (\forall n : \mathbf{nat}, P \ n \rightarrow P \ (S \ n)) \rightarrow \forall n : \mathbf{nat}, P \ n$$

Theorem `n_plus_0'` : $\forall n : \mathbf{nat}, \text{plus } n \ 0 = n$.

```

  apply (nat_ind (fun n => plus n 0 = 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 0 → (∀ 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 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

```

Check *nat_rect*.

```

nat_rect
  : ∀ P : nat → Type,
    P 0 → (∀ 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 0 → (∀ 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 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

```

Check *nat_rect*.

```

nat_rect
  : ∀ P : nat → Type,
    P 0 → (∀ 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 0 → (∀ 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 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

```

Check *nat_rect*.

```

nat_rect
  : ∀ P : nat → Type,
    P 0 → (∀ 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 $\{T\}$ ($ls : \text{list } T$) : nat :=
 match ls with
 | Nil \Rightarrow 0
 | Cons _ ls' \Rightarrow S (length ls')
 end.

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

Theorem length_app : $\forall T\ (ls1\ ls2 : \text{list } T)$, length (app $ls1\ ls2$)
 = plus (length $ls1$) (length $ls2$).
 induction $ls1$...

Qed.

Parametric lists cont.

Arguments Nil $\{T\}$.

Arguments Cons $[T]$.

Fixpoint length $\{T\} (ls : \mathbf{list} T) : \mathbf{nat} :=$
 match ls with
 | Nil \Rightarrow 0
 | Cons _ $ls' \Rightarrow$ S (length ls')
 end.

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

Theorem length_app : $\forall T (ls1 ls2 : \mathbf{list} T),$ length (app $ls1 ls2$)
 = plus (length $ls1$) (length $ls2$).
 induction $ls1$...

Qed.

Parametric lists cont.

Arguments Nil { T }.

Arguments Cons [T].

Fixpoint length { T } ($ls : \mathbf{list} T$) : **nat** :=
 match ls with
 | Nil \Rightarrow 0
 | Cons _ ls' \Rightarrow S (length ls')
 end.

Fixpoint app { T } ($ls1\ ls2 : \mathbf{list} T$) : **list** T :=
 match $ls1$ with
 | Nil \Rightarrow $ls2$
 | Cons $x\ ls1'$ \Rightarrow Cons x (app $ls1'\ ls2$)
 end.

Theorem length_app : $\forall T (ls1\ ls2 : \mathbf{list} T), \text{length (app } ls1\ ls2)$
 = plus (length $ls1$) (length $ls2$).
 induction $ls1$...

Qed.

Parametric lists cont.

Arguments Nil $\{T\}$.

Arguments Cons $[T]$.

Fixpoint length $\{T\}$ ($ls : \mathbf{list}\ T$) : \mathbf{nat} :=
 match ls with
 | Nil \Rightarrow 0
 | Cons _ ls' \Rightarrow S (length ls')
 end.

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

Theorem length_app : $\forall T (ls1\ ls2 : \mathbf{list}\ T), \text{length (app } ls1\ ls2)$
 = 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
  :  $\forall (P : \mathbf{even\_list} \rightarrow \text{Prop}) (P0 : \mathbf{odd\_list} \rightarrow \text{Prop}),$ 
     $P \text{ ENil} \rightarrow$ 
     $(\forall (n : \mathbf{nat}) (o : \mathbf{odd\_list}), P0\ o \rightarrow P (\text{ECons } n\ o)) \rightarrow$ 
     $(\forall (n : \mathbf{nat}) (e : \mathbf{even\_list}), P\ e \rightarrow P0 (\text{OCons } n\ e)) \rightarrow$ 
     $\forall e : \mathbf{even\_list}, P\ e$ 
  
```

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
  :  $\forall (P : \mathbf{even\_list} \rightarrow \text{Prop}) (P0 : \mathbf{odd\_list} \rightarrow \text{Prop}),$ 
     $P \text{ ENil} \rightarrow$ 
     $(\forall (n : \mathbf{nat}) (o : \mathbf{odd\_list}), P0\ o \rightarrow P (\text{ECons } n\ o)) \rightarrow$ 
     $(\forall (n : \mathbf{nat}) (e : \mathbf{even\_list}), P\ e \rightarrow P0 (\text{OCons } n\ e)) \rightarrow$ 
     $\forall e : \mathbf{even\_list}, P\ e$ 

```

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)

$$\text{exType} : (P : \text{Type}_i \rightarrow \text{Prop}) \rightarrow \text{Type}_j$$

$$\text{exT_intro} : \forall X : \text{Type}_k, P X \rightarrow \text{exType } P$$

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)

$$\text{exType} : (P : \text{Type}_i \rightarrow \text{Prop}) \rightarrow \text{Type}_j$$

$$\text{exT_intro} : \forall X : \text{Type}_k, P X \rightarrow \text{exType } P$$

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)

$$\text{exType} : (P : \text{Type}_i \rightarrow \text{Prop}) \rightarrow \text{Type}_j$$

$$\text{exT_intro} : \forall X : \text{Type}_k, P X \rightarrow \text{exType } P$$

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 : 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 : 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 : 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
```