

# Typowanie i typy indukcyjne w Coqu

Daria Walukiewicz-Chrząszcz

14 kwietnia 2017

# Coq — reguły typowania: środowisko

środowisko  $\vdash$  term : typ

środowisko: globalne i lokalne deklaracje i definicje

W manualu Coqa są:

- 4 reguły na rozszerzanie środowiska globalnego/lokalnego o definicje/deklaracje
- 2 reguły dostępu do stałych/zmiennych ze środowiska globalnego/lokalnego

# Coq — reguły typowania: środowisko

środowisko  $\vdash$  term : typ

środowisko: globalne i lokalne deklaracje i definicje

W manualu Coqa są:

- 4 reguły na rozszerzanie środowiska globalnego/lokalnego o definicje/deklaracje
- 2 reguły dostępu do stałych/zmiennych ze środowiska globalnego/lokalnego

# Coq — reguły typowania: środowisko

$\text{środowisko} \vdash \text{term} : \text{typ}$

środowisko: globalne i lokalne deklaracje i definicje

W manualu Coqa są:

- 4 reguły na rozszerzanie środowiska globalnego/lokalnego o definicje/deklaracje
- 2 reguły dostępu do stałych/zmiennych ze środowiska globalnego/lokalnego

# Coq — reguły typowania: środowisko

$\text{środowisko} \vdash \text{term} : \text{typ}$

środowisko: globalne i lokalne deklaracje i definicje

W manualu Coqa są:

- 4 reguły na rozszerzanie środowiska globalnego/lokalnego o definicje/deklaracje
- 2 reguły dostępu do stałych/zmiennych ze środowiska globalnego/lokalnego

# Coq — reguły typowania: sorty

- Sorty w Coqu:

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

- Kumulatywność (lub podtypowanie):

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

# Coq — reguły typowania: sorty

- Sorty w Coqu:

$$\begin{array}{l} Prop \\ Set \end{array} : Type(1): Type(2): \dots$$

- Kumulatywność (lub podtypowanie):

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

## Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów zależnych:

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

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

reguła aplikacji:

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



## Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów zależnych:

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

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

reguła aplikacji:

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

## Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów zależnych:

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

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

reguła aplikacji:

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

## Coq — reguły typowania: produkty

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

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

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

## Coq — redukcje

## • beta

$$(\lambda x:A.M)N \longrightarrow_{\beta} M[N/x]$$

## • eta

$$M \longrightarrow_{\eta} \lambda x:A.Mx \quad \text{jesli } M \text{ jest typu funkcyjnego}$$

## • delta

(rozwijanie definicji)

## • zeta

$$(\text{let } x:=N \text{ in } M) \longrightarrow_{\zeta} M[N/x]$$

## • iota

(związane z typami indukcyjnymi)

## Coq — redukcje

- beta

$$(\lambda x:A.M)N \longrightarrow_{\beta} M[N/x]$$

- eta

$$M \longrightarrow_{\eta} \lambda x:A.Mx \quad \text{jesli } M \text{ jest typu funkcyjnego}$$

- delta

(rozwijanie definicji)

- zeta

$$(\text{let } x:=N \text{ in } M) \longrightarrow_{\zeta} M[N/x]$$

- iota

(związane z typami indukcyjnymi)

## Coq — redukcje

- beta

$$(\lambda x:A.M)N \longrightarrow_{\beta} M[N/x]$$

- eta

$$M \longrightarrow_{\eta} \lambda x:A.Mx \quad \text{jesli } M \text{ jest typu funkcyjnego}$$

- delta

(rozwijanie definicji)

- zeta

$$(\text{let } x:=N \text{ in } M) \longrightarrow_{\zeta} M[N/x]$$

- iota

(związane z typami indukcyjnymi)

## Coq — redukcje

- beta

$$(\lambda x:A.M)N \longrightarrow_{\beta} M[N/x]$$

- eta

$$M \longrightarrow_{\eta} \lambda x:A.Mx \quad \text{jesli } M \text{ jest typu funkcyjnego}$$

- delta

(rozwijanie definicji)

- zeta

$$(\text{let } x:=N \text{ in } M) \longrightarrow_{\zeta} M[N/x]$$

- iota

(związane z typami indukcyjnymi)

## Coq — redukcje

- beta

$$(\lambda x:A.M)N \longrightarrow_{\beta} M[N/x]$$

- eta

$$M \longrightarrow_{\eta} \lambda x:A.Mx \quad \text{jesli } M \text{ jest typu funkcyjnego}$$

- delta

(rozwijanie definicji)

- zeta

$$(\text{let } x:=N \text{ in } M) \longrightarrow_{\zeta} M[N/x]$$

- iota

(związane z typami indukcyjnymi)



## Coq — konwersja

reguła konwersji

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \quad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

vector nat 4 =<sub>iota</sub> vector nat (2+2)

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ jeśli } s_1 \leq s_2$$

## Coq — konwersja

reguła konwersji

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \quad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

vector nat 4 =<sub>iota</sub> vector nat (2+2)

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ jeśli } s_1 \leq s_2$$

## Coq — konwersja

reguła konwersji

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \quad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

$$\text{vector nat } 4 =_{\text{iota}} \text{vector nat } (2+2)$$

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ jeśli } s_1 \leq s_2$$

## Coq — konwersja

reguła konwersji

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \quad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

vector nat 4 =<sub>iota</sub> vector nat (2+2)

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ jeśli } s_1 \leq s_2$$

## Coq — produkty

- typ funkcyjny ( $\lambda n : \text{nat}.n + 1$ )

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

- polimorfizm ( $\lambda \alpha : \text{Set}.\lambda x : \alpha.x$ )

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

- typ predykatu (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 — produkty

- typ funkcyjny ( $\lambda n : \text{nat}.n + 1$ )

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

- polimorfizm ( $\lambda \alpha : \text{Set}.\lambda x : \alpha.x$ )

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

- typ predykatu (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 — produkty

- typ funkcyjny ( $\lambda n : \text{nat}.n + 1$ )

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

- polimorfizm ( $\lambda \alpha : \text{Set}.\lambda x : \alpha.x$ )

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

- typ predykatu (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 — produkty (cd.)

- konstruktor typu (List)

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

- typ zależny (ftree)

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

- impredykatywność (numerały Churcha): użyta reguła **Prod-Prop**

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



## Coq — produkty (cd.)

- konstruktor typu (List)

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

- typ zależny (ftree)

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

- impredykatywność (numerały Churcha): użyta reguła **Prod-Prop**

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

## Coq — produkty (cd.)

- konstruktor typu (List)

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

- typ zależny (ftree)

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

- impredykatywność (numerały Churcha): użyta reguła **Prod-Prop**

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

## Liczby naturalne (pattern matching, rekursja, iota)

```

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.

```

## Liczby naturalne (pattern matching, rekursja, iota)

```
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.
```

## Liczby naturalne (pattern matching, rekursja, iota)

```
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.
```

## Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

`plus 0 0 = 0`

`reflexivity`.

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl`.

`rewrite IHn`.

`reflexivity`.

`Qed`.

## Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

`plus 0 0 = 0`  
*reflexivity*.

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

`n : nat`  
`IHn : plus n 0 = n`  
 =====  
`plus (S n) 0 = S n`

*simpl.*  
*rewrite IHn.*  
*reflexivity.*  
*Qed.*

## Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

`plus 0 0 = 0`

`reflexivity`.

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl.`

`rewrite IHn.`

`reflexivity.`

`Qed.`



## Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

`plus 0 0 = 0`

`reflexivity`.

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl.`

`rewrite IHn.`

`reflexivity.`

`Qed.`

## Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

`plus 0 0 = 0`

`reflexivity`.

jest prawdziwy z powodu konwersji (iota reguła). Drugi cel dowodowy to:

`n : nat`

`IHn : plus n 0 = n`

=====

`plus (S n) 0 = S n`

`simpl.`

`rewrite IHn.`

`reflexivity.`

`Qed.`

## Liczby naturalne — zasada indukcji

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.
```

## Liczby naturalne — zasada indukcji

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.
```

## Najprostsze typy indukcyjne

```
Inductive unit : Set :=
  | tt.
```

Check unit\_ind.

```
unit_ind :  $\forall P : \mathbf{unit} \rightarrow \text{Prop}, P \text{ tt} \rightarrow \forall u : \mathbf{unit}, P u$ 
```

```
Inductive Empty_set : Set := .
```

Check Empty\_set\_ind.

```
Empty_set_ind :  $\forall (P : \mathbf{Empty\_set} \rightarrow \text{Prop}) (e : \mathbf{Empty\_set}), P e$ 
```

## Najprostsze typy indukcyjne

```
Inductive unit : Set :=
  | tt.
```

```
Check unit_ind.
```

```
unit_ind :  $\forall P : \mathbf{unit} \rightarrow \text{Prop}, P \text{ tt} \rightarrow \forall u : \mathbf{unit}, P u$ 
```

```
Inductive Empty_set : Set := .
```

```
Check Empty_set_ind.
```

```
Empty_set_ind :  $\forall (P : \mathbf{Empty\_set} \rightarrow \text{Prop}) (e : \mathbf{Empty\_set}), P e$ 
```

## Najprostsze typy indukcyjne

Inductive **unit** : Set :=  
 | tt.

Check unit\_ind.

unit\_ind :  $\forall P : \mathbf{unit} \rightarrow \text{Prop}, P \text{ tt} \rightarrow \forall u : \mathbf{unit}, P u$

Inductive **Empty\_set** : Set := .

Check Empty\_set\_ind.

Empty\_set\_ind :  $\forall (P : \mathbf{Empty\_set} \rightarrow \text{Prop}) (e : \mathbf{Empty\_set}), P e$

## Najprostsze typy indukcyjne

Inductive **unit** : Set :=  
 | tt.

Check unit\_ind.

unit\_ind :  $\forall P : \mathbf{unit} \rightarrow \text{Prop}, P \text{ tt} \rightarrow \forall u : \mathbf{unit}, P u$

Inductive **Empty\_set** : Set := .

Check Empty\_set\_ind.

Empty\_set\_ind :  $\forall (P : \mathbf{Empty\_set} \rightarrow \text{Prop}) (e : \mathbf{Empty\_set}), P e$



## Drzewa

```

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

```

## Drzewa

```

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

```

## Typy indukcyjne z parametrami: list

```

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

```

```

Fixpoint length T (ls : list T) : nat :=
  match ls with
  | Nil ⇒ 0
  | 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.

## Typy indukcyjne z parametrami: list

```

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

```

```

Fixpoint length T (ls : list T) : nat :=
  match ls with
  | Nil ⇒ 0
  | 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.

## Typy indukcyjne z parametrami: list

```

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

```

```

Fixpoint length T (ls : list T) : nat :=
  match ls with
  | Nil ⇒ 0
  | 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.

## Typy indukcyjne z parametrami: list

```

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

```

```

Fixpoint length T (ls : list T) : nat :=
  match ls with
  | Nil ⇒ 0
  | 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.

## Listy z parametrem — zasada indukcji

Check list\_ind.

list\_ind

$$\begin{aligned}
 &: \forall (T : \text{Set}) (P : \text{list } T \rightarrow \text{Prop}), \\
 &\quad P (\text{Nil } T) \rightarrow \\
 &\quad (\forall (t : T) (l : \text{list } T), P l \rightarrow P (\text{Cons } t l)) \rightarrow \\
 &\quad \forall l : \text{list } T, P l
 \end{aligned}$$

## Listy bez parametru — typ i zasada indukcji

```

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

```



## Listy bez parametru — typ i zasada indukcji

```

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

```

## Listy bez parametru — typ i zasada indukcji

```

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

```

Typy indukcyjne wzajemnie rekurencyjne: `odd_list`, `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
```

Typy indukcyjne wzajemnie rekurencyjne: `odd_list`, `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 - generowanie zasad indukcji

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 - generowanie zasad indukcji

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

## Typy indukcyjne funkcyjne (reflexive types): pformula

```

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

```

## Typy indukcyjne funkcyjne (reflexive types): pformula

```

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

```



## Zasady indukcji i rekursory

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

```

## Zasady indukcji i rekursory

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

```

## Zasady indukcji i rekursory

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

```

W Coqu `nat_rect` definiowane przez `fix` i `match`

Print `nat_rect`.

```

nat_rect =
fun (P : nat → Type) (f : P 0) (f0 : ∀ n : nat, P n → P (S n)) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 ⇒ f
  | S n' ⇒ f0 n' (F n')
end
: ∀ P : nat → Type,
  P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n

```

## Zagnieżdżone typy indukcyjne: nat\_tree

```
Inductive nat_tree : Set :=
| NNode' : nat → list nat_tree → nat_tree.
```

Automatycznie generowana zasada indukcji jest “nieużywalna”  
 (“używalną” trzeba udowodnić samemu)

Check nat\_tree\_ind.

```
nat_tree_ind
: ∀ P : nat_tree → Prop,
  (∀ (n : nat) (l : list nat_tree), P (NNode' n l)) →
  ∀ n : nat_tree, P n
```

## Zagnieżdżone typy indukcyjne: nat\_tree

Inductive **nat\_tree** : Set :=  
 | NNode' : **nat** → **list nat\_tree** → **nat\_tree**.

Automatycznie generowana zasada indukcji jest “nieużywalna”  
 (“używalną” trzeba udowodnić samemu)

Check nat\_tree\_ind.

nat\_tree\_ind  
 :  $\forall P : \mathbf{nat\_tree} \rightarrow \mathbf{Prop},$   
    $(\forall (n : \mathbf{nat}) (l : \mathbf{list\ nat\_tree}), P (\mathbf{NNode}'\ n\ l)) \rightarrow$   
    $\forall n : \mathbf{nat\_tree}, P\ n$

# Ograniczenia: positivity condition

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

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

# Ograniczenia: positivity condition

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

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



## Ograniczenia: w Set są tylko small inductive types

Poprawna (definicja formuły  $\exists \phi P(\phi)$ ):

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

Niepoprawna:

```
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.

Poprawna:

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

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

pod warunkiem, że  $X$  typu  $\text{Type}_k$  takiego że  $k < j$  oraz  $k \leq i$

## Ograniczenia: w Set są tylko small inductive types

Poprawna (definicja formuły  $\exists \phi P(\phi)$ ):

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

Niepoprawna:

```
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.

Poprawna:

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

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

pod warunkiem, że  $X$  typu  $\text{Type}_k$  takiego że  $k < j$  oraz  $k \leq i$

## Ograniczenia: w Set są tylko small inductive types

Poprawna (definicja formuły  $\exists \phi P(\phi)$ ):

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

Niepoprawna:

```
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.

Poprawna:

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

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

pod warunkiem, że  $X$  typu  $\text{Type}_k$  takiego że  $k < j$  oraz  $k \leq i$

# Predykaty rekurencyjne

```

Inductive even : nat → Prop :=
| EvenO : even 0
| EvenSS : ∀ n, even n → even (S (S n)).

```

Theorem even\_plus :  $\forall n m, \text{even } n \rightarrow \text{even } m \rightarrow \text{even } (n + m)$ .

induction 1.

$m : \text{nat}$

=====

$\text{even } m \rightarrow \text{even } (0 + m)$

# Predykaty rekurencyjne

```

Inductive even : nat → Prop :=
| EvenO : even 0
| EvenSS : ∀ n, even n → even (S (S n)).

```

Theorem even\_plus :  $\forall n m, \text{even } n \rightarrow \text{even } m \rightarrow \text{even } (n + m)$ .

induction 1.

$m : \text{nat}$

=====

$\text{even } m \rightarrow \text{even } (0 + m)$

# Predykaty rekurencyjne

```

Inductive even : nat → Prop :=
| EvenO : even 0
| EvenSS : ∀ n, even n → even (S (S n)).

```

Theorem even\_plus :  $\forall n m, \text{even } n \rightarrow \text{even } m \rightarrow \text{even } (n + m)$ .

induction 1.

$m : \text{nat}$

```

=====
even m → even (0 + m)

```

# Predykaty rekurencyjne

```

Inductive even : nat → Prop :=
| EvenO : even 0
| EvenSS : ∀ n, even n → even (S (S n)).

```

Theorem even\_plus :  $\forall n m, \text{even } n \rightarrow \text{even } m \rightarrow \text{even } (n + m)$ .

induction 1.

$m : \text{nat}$

```

=====
even m → even (0 + m)

```

## Dowód indukcyjny dla Even c.d.

subgoal 2 is:

**even**  $m \rightarrow$  **even** (**S** (**S**  $n$ ) +  $m$ )

intro.

$m$  : **nat**

$n$  : **nat**

$H$  : **even**  $n$

$IHeven$  : **even**  $m \rightarrow$  **even** ( $n + m$ )

$H0$  : **even**  $m$

=====

**even** (**S** (**S**  $n$ ) +  $m$ )

simpl; constructor.



## Dowód indukcyjny dla Even c.d.

subgoal 2 is:

**even**  $m \rightarrow$  **even** (**S** (**S**  $n$ ) +  $m$ )

intro.

$m$  : **nat**

$n$  : **nat**

$H$  : **even**  $n$

$IHeven$  : **even**  $m \rightarrow$  **even** ( $n + m$ )

$H0$  : **even**  $m$

=====

**even** (**S** (**S**  $n$ ) +  $m$ )

simpl; constructor.

## Dowód indukcyjny dla Even c.d.

subgoal 2 is:

**even**  $m \rightarrow$  **even** (**S** (**S**  $n$ ) +  $m$ )

intro.

$m$  : **nat**

$n$  : **nat**

$H$  : **even**  $n$

$IHeven$  : **even**  $m \rightarrow$  **even** ( $n + m$ )

$H0$  : **even**  $m$

=====

**even** (**S** (**S**  $n$ ) +  $m$ )

simpl; constructor.

## Dowód indukcyjny dla Even c.d.

$m : \mathbf{nat}$

$n : \mathbf{nat}$

$H : \mathbf{even} \ n$

$IHeven : \mathbf{even} \ m \rightarrow \mathbf{even} \ (n + m)$

$H0 : \mathbf{even} \ m$

=====

$\mathbf{even} \ (n + m)$

`apply IHeven; assumption.`

## Dowód indukcyjny dla Even c.d.

 $m : \mathbf{nat}$  $n : \mathbf{nat}$  $H : \mathbf{even} \ n$  $IHeven : \mathbf{even} \ m \rightarrow \mathbf{even} \ (n + m)$  $H0 : \mathbf{even} \ m$ 

```
=====
even (n + m)
```

```
apply IHeven; assumption.
```

## Równość definiowalna

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

```
Print "=".
```

```
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
```

## Równość definiowalna

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

```
Print "=".
```

```
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
```

## Równość definiowalna

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

```
Print "=".
```

```
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
```

## Równość definiowalna

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

```
Print "=".
```

```
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
```



## Taktyki reflexivity i rewrite

```
reflexivity ≡ apply eq_refl
```

```
rewrite H dla H: a=b ≡ apply eq_ind dla odpowiedniego P
```

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

## Taktyki reflexivity i rewrite

reflexivity  $\equiv$  apply eq\_refl

rewrite H dla H: a=b  $\equiv$  apply eq\_ind dla odpowiedniego P

eq\_ind: forall (A : Type) (x : A) (P : A -> Prop),  
                                   P x -> forall y : A, x = y -> P y

# Prop to nie bool !

- W pustym środowisku programy typu bool zawsze obliczają się do `true` albo `false`
- predykaty logiczne mogą wyrażać własności nierozstrzygalne, np własność stopu danej maszyny Turinga na danym wejściu
- dlatego predykaty nie mogą "być w" `bool`.

# Prop to nie bool !

- W pustym środowisku programy typu bool zawsze obliczają się do `true` albo `false`
- predykaty logiczne mogą wyrażać własności nierozstrzygalne, np własność stopu danej maszyny Turinga na danym wejściu
- dlatego predykaty nie mogą "być w" `bool`.

# Prop to nie bool !

- W pustym środowisku programy typu bool zawsze obliczają się do `true` albo `false`
- predykaty logiczne mogą wyrażać własności nierozstrzygalne, np własność stopu danej maszyny Turinga na danym wejściu
- dlatego predykaty nie mogą “być w” `bool`.