# Typowanie i typy indukcyjne w Coqu

Daria Walukiewicz-Chrząszcz

27 marca 2018

# Coq — reguły typowania: środowisko

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

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

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

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

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

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

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

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:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type(1) : Type(2) : \ldots$$

- Kumulatywność (lub podtypowanie):

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

# Coq — reguły typowania: sorty

- Sorty w Coqu:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type(1) : Type(2) : \ldots$$

- Kumulatywność (lub podtypowanie):

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

# Coq — sorty Prop i Set

- `A` jest formułą, zdaniem logicznym, `d` jest dowodem `A`

$$d : A : Prop$$

- `T` jest typem, `n` jest wartością typu `T`

$$n : T : Set$$

# Coq — sorty Prop i Set

- `A` jest formułą, zdaniem logicznym, `d` jest dowodem `A`

$$d : A : Prop$$

- `T` jest typem, `n` jest wartością typu `T`

$$n : T : Set$$

# Coq — sorty Prop i Set

- `A` jest formułą, zdaniem logicznym, `d` jest dowodem `A`

$$d : A : Prop$$

- `T` jest typem, `n` jest wartością typu `T`

$$n : T : Set$$

# 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: $A \to B$ 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: 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: $A \to B$ 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: 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: $A \to B$ 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

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

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

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

`Prop` jest impredykatywny, `Set`, `Type` są predykatywne

# Coq — reguły typowania: produkty

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

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

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

`Prop` jest impredykatywny, `Set`, `Type` są predykatywne

# Coq — redukcje

- beta

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

- eta

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

- delta

  (rozwijanie definicji)

- zeta

  `(let x:=N 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

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

- delta

  (rozwijanie definicji)

- zeta

  `(let x:=N 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

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

- delta

  (rozwijanie definicji)

- zeta

  `(let x:=N 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

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

- delta

  (rozwijanie definicji)

- zeta

  `(let x:=N 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

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

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

# Coq — konwersja

reguła konwersji

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

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

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

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

# Coq — konwersja

reguła konwersji

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

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

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

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

# Coq — konwersja

reguła konwersji

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

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

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

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

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

tak naprawdę rozszerzona o podtypowanie na sortach, tak jakby:

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

# Coq — produkty

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

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

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

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

- typ predykatu $(\texttt{Even})$

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

# Coq — produkty

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

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

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

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

- typ predykatu (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 — produkty

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

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

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

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

- typ predykatu (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 — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`ftree`)

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

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

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

# Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`ftree`)

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

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

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

# Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`ftree`)

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

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

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

# 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_O : ∀ n : nat, plus n O = n.
  induction n.
```

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

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

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

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

# Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

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

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

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

# Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

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

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

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

# Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

plus O O = O

```
  reflexivity.
```

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

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

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

# Liczby naturalne — indukcja

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

Pierwszy cel dowodowy to:

```
plus O O = O
```

```
  reflexivity.
```

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

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

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

# Liczby naturalne — zasada indukcji

```
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.
```

# Liczby naturalne — zasada indukcji

```
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.
```

# Najprostsze typy indukcyjne

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

Check unit_ind.
```

unit_ind : ∀ $P$ : **unit** → Prop, $P$ tt → ∀ $u$ : **unit**, $P$ $u$

```
Inductive Empty_set : Set := .

Check Empty_set_ind.
```

Empty_set_ind : ∀ ($P$ : **Empty_set** → Prop) ($e$ : **Empty_set**), $P$ $e$

# Najprostsze typy indukcyjne

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

Check unit_ind.
```

unit_ind : $\forall\ P : \mathbf{unit} \to \mathtt{Prop},\ P\ \mathtt{tt} \to \forall\ u : \mathbf{unit},\ P\ u$

```
Inductive Empty_set : Set := .

Check Empty_set_ind.
```

Empty_set_ind : $\forall\ (P : \mathbf{Empty_set} \to \mathtt{Prop})\ (e : \mathbf{Empty_set}),\ P\ e$

# Najprostsze typy indukcyjne

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

Check unit_ind.
```

unit_ind : $\forall$ $P$ : **unit** $\rightarrow$ Prop, $P$ tt $\rightarrow$ $\forall$ $u$ : **unit**, $P$ $u$

```
Inductive Empty_set : Set := .

Check Empty_set_ind.
```

Empty_set_ind : $\forall$ ($P$ : **Empty_set** $\rightarrow$ Prop) ($e$ : **Empty_set**), $P$ $e$

# Najprostsze typy indukcyjne

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

Check unit_ind.
```

unit_ind : $\forall$ $P$ : **unit** $\rightarrow$ Prop, $P$ tt $\rightarrow$ $\forall$ $u$ : **unit**, $P$ $u$

```
Inductive Empty_set : Set := .

Check Empty_set_ind.
```

Empty_set_ind : $\forall$ ($P$ : **Empty_set** $\rightarrow$ Prop) ($e$ : **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 (przy Set Implicit Arguments)

```
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 ⇒ 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.
```

# Typy indukcyjne z parametrami: list (przy Set Implicit Arguments)

```
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 ⇒ 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.
```

# Typy indukcyjne z parametrami: list (przy Set Implicit Arguments)

```
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 ⇒ 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.
```

# Listy z parametrem — zasada indukcji

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

```
Check list_ind.

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

# Listy z parametrem — zasada indukcji

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

```
Check list_ind.

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

# 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
     : ∀ (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 - 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
     : ∀ (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
```

# 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 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
```

# Zasady indukcji i rekursory

```
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
```

# Zasady indukcji i rekursory

```
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
```

# W Coqu nat_rect definiowane przez fix i 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
```

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

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

pod warunkiem, że $X$ typu $\mathtt{Type}_k$ 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.
```

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

pod warunkiem, że $X$ typu $\mathtt{Type}_k$ 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.
```

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

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

# Predykaty rekurencyjne

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

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

# Predykaty rekurencyjne

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

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

# Predykaty rekurencyjne

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

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

# Predykaty rekurencyjne

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

```
Theorem even_plus : ∀ n m, even n → even m → even (n + m).
```

```
  induction 1.
```

```
  m : nat
  ============================
   even m → even (0 + m)
```

# Dowód indukcyjny dla Even c.d.

`subgoal 2 is:`

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

```
intro.
```

$m$ : **nat**
$n$ : **nat**
$H$ : **even** $n$
$IHeven$ : **even** $m \to$ **even** $(n + m)$
$H0$ : **even** $m$
==========
**even** (S (S $n$) + $m$)

```
simpl; constructor.
```

# Dowód indukcyjny dla Even c.d.

`subgoal 2 is:`

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

```
intro.
```

$m$ : **nat**
$n$ : **nat**
$H$ : **even** $n$
$IHeven$ : **even** $m \to$ **even** $(n + m)$
$H0$ : **even** $m$
==========
**even** (S (S $n$) + $m$)

```
simpl; constructor.
```

# Dowód indukcyjny dla Even c.d.

`subgoal 2 is:`

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

```
intro.
```

$m$ : **nat**
$n$ : **nat**
$H$ : **even** $n$
$IHeven$ : **even** $m \to$ **even** $(n + m)$
$H0$ : **even** $m$
==========
**even** (S (S $n$) + $m$)

```
simpl; constructor.
```

# Dowód indukcyjny dla Even c.d.

$m$ : **nat**
$n$ : **nat**
$H$ : **even** $n$
$IHeven$ : **even** $m \rightarrow$ **even** $(n + m)$
$H0$ : **even** $m$
==========
**even** $(n + m)$

```
apply IHeven; assumption.
```

# Dowód indukcyjny dla Even c.d.

$m$ : **nat**
$n$ : **nat**
$H$ : **even** $n$
$IHeven$ : **even** $m \to$ **even** $(n + m)$
$H0$ : **even** $m$
==========
**even** $(n + m)$

`apply` *IHeven*; `assumption`.

# Równość definiowalna

$\texttt{Inductive}\ \mathbf{eq}\ (A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \text{eq_refl} : x = x$

```
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

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

```
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** $(A : \mathtt{Type})\ (x : A) : A \to \mathtt{Prop} := \mathtt{eq_refl} : x = x$

```
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** $(A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \texttt{eq_refl} : x = x$

```
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` $\equiv$ `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
```

# 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`.