Programowanie z typami zależnymi i dowodzenie twierdzeń

Typowanie i typy indukcyjne

10.03.2015

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

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, `Set:Type`$_1$

ale nie wszystko jest typem np. `fun n:nat => n`

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

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, `Set:Type`$_1$

ale nie wszystko jest typem np. `fun n:nat => n`

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

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, `Set:Type`$_1$

ale nie wszystko jest typem np. `fun n:nat => n`

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

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, `Set:Type`$_1$

ale nie wszystko jest typem np. `fun n:nat => n`

Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

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 $x \notin FV(B)$

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{vector}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, vector n
```

reguła aplikacji:

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

Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

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 $x \notin FV(B)$

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{vector}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, vector n
```

reguła aplikacji:

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

Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

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 $x \notin FV(B)$

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{vector}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, vector n
```

reguła aplikacji:

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

Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

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 $x \notin FV(B)$

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{vector}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, vector n
```

reguła aplikacji:

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

Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

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 $x \notin FV(B)$

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{vector}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, vector n
```

reguła aplikacji:

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

Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

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 $x \notin FV(B)$

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{vector}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, vector n
```

reguła aplikacji:

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

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:

$$\begin{matrix} \mathit{Prop} \\ \mathit{Set} \end{matrix} : \mathit{Type}_1 : \mathit{Type}_2 : \ldots$$

- Kumulatywność (lub podtypowanie):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type}_1 \leq \mathit{Type}_2 \leq \ldots$$

Coq — produkty

reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają . . .

- $s_1 \leq s_2$, or
- $s_2 = Prop$

reguła podtypowania

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

Coq — produkty

reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają . . .

- $s_1 \leq s_2$, or
- $s_2 = \mathit{Prop}$

reguła podtypowania

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

Coq — produkty

reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli s_1 i s_2 spełniają . . .

- $s_1 \leq s_2$, or
- $s_2 = \mathit{Prop}$

reguła podtypowania

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

Coq — produkty

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

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

- polimorfizm $(\lambda x.x)$

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

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

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

Coq — produkty

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

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

- polimorfizm $(\lambda x.x)$

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

- typ predykatu (Even)

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

Coq — produkty

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

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

- polimorfizm $(\lambda x.x)$

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

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

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

Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`vector`)

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

- impredykatywność (numerały Churcha)

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

Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`vector`)

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

- impredykatywność (numerały Churcha)

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

Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`vector`)

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

- impredykatywność (numerały Churcha)

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

Coq — redukcje

- beta

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

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- 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

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- 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

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- 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

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- 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

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

Coq — konwersja

reguła konwersji

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

`(fun l : tstack => vstack l) nil` $=_{\beta\delta}$ `unit`

`vector 4` $=_{iota}$ `vector (2+2)`

Coq — konwersja

reguła konwersji

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

$$\texttt{(fun l : tstack => vstack l) nil} =_{\beta\delta} \texttt{unit}$$

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

Coq — konwersja

reguła konwersji

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

$$\texttt{(fun l : tstack => vstack l) nil} =_{\beta\delta} \texttt{unit}$$

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

Coq — konwersja

reguła konwersji

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

$$\texttt{(fun l : tstack => vstack l) nil} =_{\beta\delta} \texttt{unit}$$

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

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; 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; 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; 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
```

```
crush.
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
```

```
crush.
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
```

```
crush.
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
```

```
crush.
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
```

crush.
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)); crush.
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)); crush.
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 : **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

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

```
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.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1; crush.
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 ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

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

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1; crush.
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 ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

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

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1; crush.
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 ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

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

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

Listy z parametrem — zasada indukcji

```
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
```

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.
```

```
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.
```

```
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
```

Własnoręczne definiowanie zasady indukcji dla nat_tree

```
Section nat_tree_ind'.
  Variable P : nat_tree → Prop.

  Hypothesis NNode'_case : ∀ (n : nat) (ls : list nat_tree),
    All P ls → P (NNode' n ls).

  Fixpoint nat_tree_ind' (tr : nat_tree) : P tr :=
    match tr with
      | NNode' n ls ⇒ NNode'_case n ls
        ((fix list_nat_tree_ind (ls : list nat_tree) : All P ls :=
          match ls with
            | Nil ⇒ I
            | Cons tr' rest ⇒ conj (nat_tree_ind' tr')
(list_nat_tree_ind rest)
          end) ls)
    end.

End nat_tree_ind'.
```

Własnoręczne definiowanie zasady indukcji dla nat_tree

```
Section nat_tree_ind'.
  Variable P : nat_tree → Prop.

  Hypothesis NNode'_case : ∀ (n : nat) (ls : list nat_tree),
    All P ls → P (NNode' n ls).

  Fixpoint nat_tree_ind' (tr : nat_tree) : P tr :=
    match tr with
      | NNode' n ls ⇒ NNode'_case n ls
        ((fix list_nat_tree_ind (ls : list nat_tree) : All P ls :=
          match ls with
            | Nil ⇒ I
            | Cons tr' rest ⇒ conj (nat_tree_ind' tr')
(list_nat_tree_ind rest)
          end) ls)
    end.

End nat_tree_ind'.
```

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: eliminacja Prop tyko do Prop

```
Inductive or (A B:Prop) : Prop :=
lintro : A -> or A B | rintro : B -> or A B.
```

Niepoprawna:

```
Definition choice (A B: Prop) (x:or A B) :=
match x with lintro a => true | rintro b => false end.
```

```
Error: Incorrect elimination of "x" in the inductive type "or":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop is not allowed
on a predicate in sort Set because proofs can be eliminated
only to build proofs.
```

WYJĄTEK: typy o jednym konstruktorze

Ograniczenia: eliminacja Prop tyko do Prop

```
Inductive or (A B:Prop) : Prop :=
lintro : A -> or A B | rintro : B -> or A B.
```

Niepoprawna:

```
Definition choice (A B: Prop) (x:or A B) :=
match x with lintro a => true | rintro b => false end.
```

```
Error: Incorrect elimination of "x" in the inductive type "or":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop is not allowed
on a predicate in sort Set because proofs can be eliminated
only to build proofs.
```

WYJĄTEK: typy o jednym konstruktorze

Ograniczenia: w Set są tylko small inductive types

Poprawna:

```
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:

```
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 $k \leq i$

Ograniczenia: w Set są tylko small inductive types

Poprawna:

```
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 $k \leq i$