

# Typy indukcyjne w Coqu

## Definicja

Oto definicja typu indukcyjnego reprezentującego polimorficzne listy z długością, tj. dla dowolnego  $A$  typu  $\text{Set}$  i dowolnego  $n$  typu  $\text{nat}$ ,  $(\text{listn } A \ n)$  oznacza typ “lista elementów typu  $A$ , długości  $n$ ”

```
Inductive listn (A:Set) : nat -> Set :=
| niln : listn A 0
| consn : A -> forall n:nat, listn A n -> listn A (S n).
```

Po wprowadzeniu tej definicji typy  $\text{listn}$  i jego konstruktorów są następujące:

```
listn : Set -> nat -> Set
niln  : forall A:Set, listn A 0
consn : forall A:Set, A-> forall n:nat, listn A n -> listn A (S n)
```

Na definicję indukcyjną składają się następujące elementy:

- nazwa typu indukcyjnego –  $\text{listn}$ ,
- lista parametrów –  $(A:\text{Set})$

Parametry to niezmiennie argumenty typu indukcyjnego. Wszystkie wystąpienia  $\text{listn}$  w „zakończeniach” typów konstruktorów muszą być zaaplikowane do parametru  $A$ . Z tego powodu kolejny argument  $\text{listn}$  typu  $\text{nat}$  nie mógłby być parametrem. Oprócz tego typy konstruktorów oraz samego  $\text{listn}$  są abstrahowane ze względu na parametry.

- typ (rodzaj) typu indukcyjnego –  $\text{nat} \rightarrow \text{Set}$

Musi to być produkt (może być zależny) zakończony jednym z sortów ( $\text{Set}$ ,  $\text{Prop}$  lub  $\text{Type}$ ). Jeśli sortem tym jest  $\text{Set}$  mówimy o *typie* indukcyjnym, jeśli  $\text{Prop}$  – o *predykcji* indukcyjnym.

- lista deklaracji konstruktorów oddzielona  $|$

Każda deklaracja konstruktora składa się z jego nazwy, oraz typu. Typ musi być produktem (może być zależnym), zakończonym definiowanym typem indukcyjnym, zaaplikowanym do parametrów i innych argumentów, zależnych od konstruktora, czyli w tym przypadku  $\text{listn } A \ \_$  gdzie ostatnia pozycja zależy od konstruktora.

Dodatkowym warunkiem, który muszą spełniać typy konstruktorów jest warunek pozytywności. Z grubsza mówi on, że jeśli  $\text{listn}$  występuje w typie *argumentu* konstruktora to występuje wyłącznie na końcu, tzn. typy argumentów konstruktora mają postać  $\text{coś}_1 \rightarrow \dots \rightarrow \text{coś}_n \rightarrow \text{listn } A \ \text{coś}$ , gdzie w  $\text{coś}_i$  oraz  $\text{coś}$  nie występuje  $\text{listn}$ . Następująca deklaracja konstruktora:

```
nbzdura : A -> forall n:nat, (listn A n -> A) -> listn A n
```

nie jest poprawna.

## Match – analiza przypadków

Założmy, że chcemy udowodnić predykat  $P$  dla dowolnej listy, wiedząc, że jest prawdziwy dla dowolnej listy pustej oraz dla dowolnej listy niepustej.

Mamy zatem:

```
A:Set
P:forall n:nat, listn A n -> Prop
d_pusta : P 0 (niln A)
d_niepusta : forall (n':nat)(a':A)(l':(list A n')), P (S n') (consn A a' n' l')
n:nat
l:list A n
=====
match l in list _ n return P n l with
  niln => d_pusta
| consn a n l => d_niepusta n a l
end
: P n l
```

Należy zwrócić uwagę na następujące fakty:

- parametr  $A$  jest globalny. Mógłby to być też jakiś konkretny term, np. typ  $\text{nat} \rightarrow \text{nat}$  lub  $\text{bool}$
- typ wyjściowy `match`'a, jest podany po słowie kluczowym `return`. Jeśli nie jest on zależny, można pominąć `return coś`
- typ wyjściowy `match`'a może być zależny od parametrów `matchowanego` typu indukcyjnego (u nas  $n$ ), oraz od samego `matchowanego` elementu (u nas  $l$ );
- dla zaznaczenia zależności od parametrów typu indukcyjnego użyto `in list _ n`; podkreślenie `_` oznacza miejsce parametru typu indukcyjnego, od którego nie może zależeć  $P$ ;
- może się zdażyć, że `matchowany` element nie jest zmienną i/lub że jego typ ma argumenty, które nie są zmiennymi; np. jeśli mamy daną funkcję `appendn`:

```
appendn: forall (A:Set) (n m:nat), listn A n -> listn A m -> listn A (n+m)
i dane dwie listy ln : list A n i lm : list A m, wówczas możemy chcieć wykonać
analizę przypadków po wyniku połączenia tych list, czyli append A n m ln lm : list
(n+m). Wówczas składnia match'a będzie następująca:
```

```
match append A n m ln lm as l in list _ k return P k l with ... end
```

konstrukcja `as l` wprowadza lokalną nazwę zmiennej dla `matchowanego` elementu, która wiąże  $l$  występujące po `return`, i podobnie `zmiana k` występujące w `list _ k` wiąże wystąpienie  $k$  po `return`.

Typy gałęzi w takim `matchu` są takie same jak w pierwszym przykładzie, a typ wynikowy, to  $P (n+m)$  (`append A n m ln lm`).

W rzeczywistości w występującej w pierwszym przykładzie konstrukcji `match l in list _ n return P n l with ... end`, `n` też jest tylko zmienną lokalną (której nazwa przypadkiem jest identyczna z globalnym `n`), natomiast `l` występuje w podwójnej roli: termu matchowanego i zmiennej wiązanej. Wewnętrznie wyrażenie to jest reprezentowane jako `match l as l' in (list _ n') return P n' l' with ... end` gdzie `l'` i `n'` to nowe zmienne.

- typy gałęzi instrukcji `match` muszą odpowiadać typom konstruktorów, tylko na końcu zamiast (`listn` parametry argumenty) stoi `P` argumenty `lc`, gdzie `lc` to lista skonstruowana danym konstruktorem.
- zauważmy, że po lewej stronie `=>` konstruktory występują bez parametru `A` (tak jak w definicji typu indukcyjnego). Po prawej muszą już mieć parametr.

Z instrukcją `match` związana jest następująca reguła, zwana  $\iota$  (jota) redukcją:

```
match niln A ... of niln => d_puste | ... end  →ι  d_puste
match consn A a' n' l' ... of ... | consn a n l => d_niepusta n a l end
                                                    →ι  d_niepuste n' a' l'
```

## Dozwolone eliminacje

W konstrukcji `match` eliminuje się obiekt typu indukcyjnego (`l : list A n`) używając predykatu eliminacyjnego (`P : forall n:nat, listn A n -> Prop`). Sortem typu indukcyjnego jest sort jakim „kończy się” jego typ; podobnie definiowany jest sort predykatu eliminacyjnego. W powyższym przykładzie sortem typu indukcyjnego jest `Set` (bo `list : forall (A:Set)(n:nat), Set`) a sortem predykatu eliminacyjnego jest `Prop` (bo `P : forall n:nat, listn A n -> Prop`).

Konstrukcja `match` nie dla wszystkich kombinacji sortów jest możliwa:

- gdy eliminujemy obiekt typu indukcyjnego z sortu `Set` lub `Type` to sort predykatu eliminacyjnego może być dowolny,
- dozwolona jest eliminacja obiektu typu indukcyjnego sortu `Prop` w sort `Prop`
- w zasadzie (patrz punkt niżej) niedozwolone są eliminacje obiektu typu indukcyjnego sortu `Prop` w sorty `Set` i `Type`, bo nie chcemy móc budować termów w tych sortach bazując na rozróżnianiu dowodów. Jednym z powodów jest ekstrakcja, np. do MLa, innym chęć używania niekiedy logiki klasycznej, która powoduje, że wszystkie dowody stają się równe (*proof irrelevance*)
- w przypadku gdy eliminujemy obiekt typu `I` sortu `Prop` i typ `I` ma zero lub jeden konstruktor to możliwe są eliminacje w dowolny sort.

## Rekurencja strukturalna

Najczęściej używana składnia operatora `fix`, to:

$$\begin{aligned} & \mathbf{fix} \ f_i \{ f_1 (x_1^1 : t_1^1) \dots (x_{i_1}^1 : t_{i_1}^1) \{ \mathbf{struct} \ x_{l_1}^1 \} : t_{k_1}^1 := d_1 \ \mathbf{with} \\ & \quad \dots \ \mathbf{with} \\ & \quad f_n (x_1^n : t_1^n) \dots (x_{i_n}^n : t_{i_n}^n) \{ \mathbf{struct} \ x_{l_n}^n \} : t_{k_n}^n := d_n \} \end{aligned}$$

Funkcje  $f_1 \dots f_n$  definiowane są wzajemnie rekurencyjnie, i następnie z tej wspólnej definicji wybierana jest funkcja  $f_i$ . Każda funkcja  $f_j$  definiowana jest przez rekurencję strukturalną ze względu na argument  $l_j$ . Typ  $t_{l_j}^j$  musi zatem być typem indukcyjnym.

W ciałach funkcji  $(d_1 \dots d_n)$  może występować dowolne  $f_j$ , ale musi być zaaplikowane do co najmniej  $l_j$  argumentów, z czego ostatni musi być *strukturalnie mniejszy* niż argument wywołania funkcji, czyli “strzeżony” poprzez co najmniej jedną konstrukcję `match`.

Powyższa składnia operatora `fix` jest równoważna następującej:

$$\begin{aligned} & \mathbf{Fix} \ f_i \{ f_1 / l_1 : \mathbf{forall} (x_1^1 : t_1^1) \dots (x_{i_1}^1 : t_{i_1}^1), t_{k_1}^1 := \mathbf{fun} (x_1^1 : t_1^1) \dots (x_{i_1}^1 : t_{i_1}^1) => d_1 \ \mathbf{with} \\ & \quad \dots \ \mathbf{with} \\ & \quad f_n / l_n : \mathbf{forall} (x_1^n : t_1^n) \dots (x_{i_n}^n : t_{i_n}^n), t_{k_n}^n := \mathbf{fun} (x_1^n : t_1^n) \dots (x_{i_n}^n : t_{i_n}^n) => d_n \} \end{aligned}$$

którą w skrócie piszemy  $\mathbf{Fix} \ f_i \{ f_1 / i_1 : A_1 := D_1 \dots f_n / i_n : A_n := D_n \}$

Reguła typowania jest zgodna z oczekiwaniem:

$$\frac{(\Gamma \vdash A_j : s_j)_{j=1 \dots n} \quad (\Gamma; f_1 : A_1 \dots f_n : A_n \vdash D_j : A_j)_{j=1 \dots n}}{\Gamma \vdash \mathbf{Fix} \ f_i \{ f_1 / i_1 : A_1 := D_1 \dots f_n / i_n : A_n := D_n \} : A_i}$$

jeśli  $s_j \in \{\mathbf{Set}, \mathbf{Prop}, \mathbf{Type}\}$  oraz oczywiście  $D_j$  są poprawnie zbudowane (zaczynają się od odpowiedniej liczby abstrakcji i przestrzegają wspomniane wyżej warunki dobrego ufundowania wywołań  $f_1 \dots f_n$ )

Reguła redukcji związana z operatorem `Fix`, zwana jest również  $\iota$ -redukcją, i wyraża się następująco (dla  $\Gamma_F \equiv f_1 / i_1 : A_1 := D_1 \dots f_n / i_n : A_n := D_n$ )

$$\mathbf{Fix} \ f_i \{ \Gamma_F \} a_1 \dots a_{i_i} \longrightarrow_{\iota} D_i \{ f_j \mapsto \mathbf{Fix} \ f_j \{ \Gamma_F \} \} a_1 \dots a_{i_i}$$

jeżeli  $a_{i_i}$  zaczyna się od konstruktora. Oczywiście bez tego warunku reguła ta w oczywisty sposób prowadziła do nieterminacji.

Żeby wyjaśnić znaczenie najczęściej używanej komendy `Fixpoint`, przyjmijmy, że

$$\begin{aligned} & \Gamma_F \equiv f_1 (x_1^1 : t_1^1) \dots (x_{i_1}^1 : t_{i_1}^1) \{ \mathbf{struct} \ x_{l_1}^1 \} : t_{k_1}^1 := d_1 \ \mathbf{with} \\ & \quad \dots \ \mathbf{with} \\ & \quad f_n (x_1^n : t_1^n) \dots (x_{i_n}^n : t_{i_n}^n) \{ \mathbf{struct} \ x_{l_n}^n \} : t_{k_n}^n := d_n \end{aligned}$$

Wprowadzenie komendy:

`Fixpoint`  $\Gamma_F$ .

odpowiada wprowadzeniu następującego ciągu komend:

`Definition`  $f_1 := \mathbf{fix} \ \Gamma_F$  `for`  $f_1$ .

...

`Definition`  $f_n := \mathbf{fix} \ \Gamma_F$  `for`  $f_n$ .

## Kilka przykładów

Zacniemy od funkcji na liczbach naturalnych (których użyjemy do definiowania funkcji na listach z długością).

```
Fixpoint npol_podloga (n:nat) : nat :=
  match n with
  | 0 => 0
  | S n' => npol_sufit n'
  end
with npol_sufit (n:nat) : nat :=
  match n with
  | 0 => 0
  | S n' => S (npol_podloga n')
  end.
```

Pierwsza z powyższych funkcji oblicza  $\lfloor \frac{n}{2} \rfloor$ , a druga  $\lceil \frac{n}{2} \rceil$ . Poniższe funkcje na listach z długością wybierają co drugi element z listy wejściowej, pierwsza z nich zaczyna od pierwszego elementu, a druga – od drugiego. Ich typy (a konkretnie długości list wyjściowych) opisane są za pomocą `npol_podloga` i `npol_sufit`.

```
Fixpoint codrugi_zpierzszym (A:Set)(n:nat)(l:listn A n) {struct l}
  : listn A (npol_sufit n) :=
  match l in listn _ n return listn A (npol_sufit n) with
  | niln => niln A
  | consn a n' l' => consn A a _ (codrugi_bezpierz A n' l')
  end
with codrugi_bezpierz (A:Set)(n:nat)(l:listn A n) {struct l}
  : listn A (npol_podloga n) :=
  match l in listn _ n return listn A (npol_podloga n) with
  | niln => niln A
  | consn a n' l' => codrugi_zpierzszym A n' l'
  end.
```

## Obiekty indukcyjne i dowodzenie

### Lematy indukcyjne

Po wprowadzeniu do środowiska definicji indukcyjnej generowane są automatycznie lematy indukcyjne. Np. Dla typu `nat` generowane są `nat_rect`, `nat_rec` i `nat_ind` o następujących typach:

```
nat_rect : forall P : nat -> Type,
  P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
nat_rec : forall P : nat -> Set,
  P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
nat_ind : forall P : nat -> Prop,
  P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Term `nat_ind` to dowód zasady indukcji na liczbach naturalnych; ogólniej `I_ind` będzie zasadą indukcji dla typu  $I$ . Dowód taki można też zbudować samemu, np

```
Fixpoint nat_ind' (P: nat->Prop)(o:P 0)(ss: forall n:nat, P n -> P (S n))
(n:nat) {struct n} : (P n) :=
  match n return P n with
  | 0 => o
  | S p => ss p (nat_rec P o ss p)
end.
```

Dla typów indukcyjnych w `Prop` generalnie będzie generowane tylko `I_ind` (dokładniej: związane to jest z tym które eliminacje są dozwolone).

Niekiedy automatycznie generowane stałe odpowiadają nieciekawym typom. Np dla typów wzajemnie rekurencyjnych `Ev` i `Odd`:

```
Inductive Ev : nat -> Prop :=
| ev0 : Ev 0
| evS : forall n:nat, Odd n -> Ev (S n)
with Odd : nat -> Prop :=
| oddS : forall n:nat, Ev n -> Odd (S n).
```

Stała `Ev_ind` ma następujący typ:

```
Ev_ind : forall P : nat -> Prop,
  P 0 ->
  (forall n : nat, Odd n -> P (S n)) -> forall n : nat, Ev n -> P n
```

Czyli dla udowodnienia  $P\ n$  dla dowolnego  $n$  spełniającego `Ev n` trzeba udowodnić  $P\ 0$  oraz pokazać, że  $P\ (S\ n)$  wynika z `Odd n` (co najczęściej będzie nie do osiągnięcia).

W takim przypadku można wygenerować potrzebne lematy indukcyjne używając konstrukcji `Scheme`, na przykład tak:

```
Scheme Ev_i := Induction for Ev Sort Prop
  with Odd_i := Induction for Odd Sort Prop.
```

Wtedy `Ev_i` ma już oczekiwany typ:

```
Ev_i : forall (P : forall n : nat, Ev n -> Prop)
  (PO : forall n : nat, Odd n -> Prop),
  P 0 ev0 ->
  (forall (n : nat) (o : Odd n), PO n o -> P (S n) (evS n o)) ->
  (forall (n : nat) (e : Ev n), P n e -> PO (S n) (oddS n e)) ->
  forall (n : nat) (e : Ev n), P n e
```

Aby użyć `Ev_i` trzeba podać „ręcznie” term `PO` używając konstrukcji `induction ... using ... with:`

```
induction H using Ev_i with (PO:= ...).
```

## Dowodzenie formuł indukcyjnych – wprowadzanie

Aby skonstruować obiekt indukcyjny z konstruktorów, używamy taktyki `constructor` *i*. np. po wprowadzeniu

```
Goal (A,B:Prop)B->A\B.  
intros.
```

mamy sytuację:

```
1 subgoal
```

```
  A : Prop  
  B : Prop  
  H : B  
  =====  
  A\B
```

Polecenie `constructor` 2. spowoduje odnalezienie drugiego konstruktora predykatu indukcyjnego `or` – i zaaplikowanie go. Jest ona zatem równoważna `apply or_intror`.

Wywołanie `constructor`. bez numeru spowoduje znalezienie pierwszego konstruktora, który może być zaaplikowany do bieżącego “celu” i zaaplikowanie go. W naszym przykładzie, byłby to konstruktor `or_introl`, co poprowadziłoby dowód na manowce.

Dla konstrukcji o dwóch konstruktorach (tak jak np. `or`, ale również np. `nat`, czy `bool`), można stosować taktyki `left.` i `right.`, które oznaczają odpowiednio `constructor` 1. i `constructor` 2.

Dla konstrukcji o jednym konstruktorze (jak `and`) taktyka `split` równoważna jest taktyce `constructor` 1.

Podobna do nich jest również taktyka `exists`, stworzona z myślą o dowodzeniu formuł egzystencjalnych (konstrukcja indukcyjna `sig` z jednym konstruktorem `exist`).

Wszystkie powyższe taktyki mogą przyjmować parametry tak samo jak taktyka `apply` (jeśli ich automatyczna synteza nie jest możliwa). Niektóre wykonują automatycznie `intros`.

## Używanie obiektów indukcyjnych – eliminacja

Pierwszą techniką eliminacji specyficzną dla obiektów indukcyjnych jest *analiza przypadków*, czyli używanie konstrukcji `match`. Służy do tego taktyka `destruct`. Tworzy ona tyle “podcelów” ile jest konstruktorów danego typu. Uwaga! Jest ona dosyć głupia – dla typów indukcyjnych zależnych bierze również pod uwagę przypadki “niemożliwe” (np. lista o długości 0 zaczynająca się od `consn`)

Wariantami `destruct` są `simple destruct` (robiące automatycznie `intros`) i taktyka bazowa `case`.

*Dowody indukcyjne* robi się w Coq’u przy pomocy taktyki `induction`. Dla danego typu indukcyjnego, np. `nat`, używa ona lematu `nat_ind`, udowodnionego automatycznie (przy pomocy `fix` i `match`) zaraz po wprowadzeniu definicji indukcyjnej. (Pozostałe stałe – `nat_rec` i `nat_rect` używane są przez `induction`, gdy “cel” jest w sorcie `Set` lub `Type`, a nie `Prop`, jak zazwyczaj).

Wariantami `induction` są `simple induction` i taktyka bazowa `elim`.

W zasadzie `elim n.` odpowiada `apply nat_ind.` o ile `nat` jest typem `n` oraz “cel” jest w sorcie `Prop`.

Do równoległych dowodów po dwóch zmiennych służy `double induction`.

Ważnymi *taktykami związanymi z równością* są `discriminate` i `injection`. Pierwsza wykorzystuje nieprawdziwe założenie o równości dwóch termów danego typu indukcyjnego zaczynających się od różnych konstruktorów do udowodnienia fałszu i w konsekwencji czegośkolwiek.

`injection` aplikuje (udowodniony naprędce) lemat o różnowartościowości konstruktorów jako funkcji.

Ich sprytnym połączeniem jest taktyka `implify_eq`.

Taktyką, która stara się nadrobić niedostatki taktyki `destruct` dla typów zależnych (jak np. nie zauważanie, że listy o długości zero nie da się skonstruować przez `consn`), jest taktyka `inversion`. Używając skomplikowanego (dowodzonego w locie) lematu wyprowadza ona jak najwięcej możliwych informacji z konkretnej instancji zależnego typu indukcyjnego.

Jej warianty to `inversion_clear` oraz `dependent inversion`.