

OBLICZENIA I WNIOSKOWANIE W SYSTEMIE COQ

Małgorzata Biernacka

Wykład 5
26.03.2013

PREDYKATY

Predykaty opisują własności obiektów:

$P(n) :=$ “ n jest liczbą pierwszą”

$Q(n, m) :=$ “ $n \leq m$ ”

$R(x) :=$ “wartość zmiennej x jest nieujemna”

- ▶ predykaty $P(n)$, $Q(n, m)$, $R(x)$ są formułami
- ▶ w Coqu predykaty traktujemy jak funkcje zwracające formuły

```
Variable P : nat -> Prop.
```

```
Variable R : var -> Prop.
```

```
Definition Q (n m : nat) := n <= m.
```

- ▶ często predykaty będziemy definiować indukcyjnie

PREDYKATY DEFINIOWANE INDUKCYJNIE

- ▶ predykaty definiowane indukcyjnie = definicje indukcyjne w Prop
- ▶ składnia analogiczna do typów indukcyjnych w Set (obowiązuje warunek pozytywnych wystąpień)
- ▶ izomorfizm Curry'ego-Howarda: dowody możemy traktować jak termy, a formuły jako typy
- ▶ konstruktory predykatów budują dowody formuł logicznych
konstruktory typów budują termy typów indukcyjnych
- ▶ w praktyce od termów (programów) i dowodów oczekujemy czegoś innego, dlatego są pewne różnice między użyciem definicji indukcyjnych w Set i Prop

STRUKTURA DOWODU NIE JEST ISTOTNA

```
Inductive mlist : Set :=  
| nil : mlist  
| cons : forall n : nat, n > 42 -> mlist -> mlist.
```

Goal

```
forall l:mlist, mmap (fun n => n) l = l.
```

- chcemy, żeby $\text{cons } n \ p \ l = \text{cons } n \ q \ l$ dla dowolnych p, q

STRUKTURA DOWODU NIE JEST ISTOTNA

- ▶ zwykle o dowodach (termach o typach formuł) chcemy wiedzieć tylko, czy istnieją, a nie – jak są zbudowane
- ▶ zasada *proof irrelevance*: dowolne dwa dowody tej samej formuły są równe
- ▶ zasada *proof irrelevance* jest niesprzeczna z pCIC, ale nie da się jej w pCIC udowodnić

```
Axiom proof_irrelevance :  
forall (A:Prop) (a b:A), a = b.
```

- ▶ przyjęcie podobnego aksjomatu w Set prowadzi do sprzeczności (a nawet słabszych jej wariantów)

SPÓJNIKI LOGICZNE DEFINIOWANE INDUKCYJNIE

- ▶ kwantyfikator ogólny i implikacja są operacjami wbudowanymi
- ▶ negacja jest standardowym skrótem
- ▶ pozostałe spójniki są definiowane indukcyjnie (możemy stosować zwykłe taktyki dla definicji indukcyjnych)

True

```
Inductive True : Prop := I : True.
```

```
Check True_ind.
```

```
> True_ind : forall P : Prop, P -> True -> P
```

- ▶ prawda konstruktywna: zawsze możemy udowodnić True podając dowód – term I
- ▶ eliminacja prawdy nie daje nam żadnych przydatnych informacji

False

Fałsz jest zdefiniowany jako indukcyjny typ pusty:

```
Inductive False : Prop :=.
```

```
Check False_ind.
```

```
> False_ind  
  : forall P:Prop, False -> P
```

- ▶ eliminacja typu indukcyjnego False to implementacja zasady *ex falso quodlibet*

KONIUNKCJA

```
Inductive and (A B:Prop) : Prop :=  
| conj : A -> B -> A /\ B.
```

```
Check and_ind.
```

```
> and_ind  
   : forall A B P:Prop, (A -> B -> P) -> A /\ B -> P
```

- ▶ wprowadzenie koniunkcji: taktyka constructor, apply conj lub split
- ▶ eliminacja koniunkcji: taktyki destruct, elim, induction, case

ALTERNATYWA

```
Inductive or (A B:Prop) : Prop :=  
| or_introl : A -> A \/ B  
| or_nintror : B -> A \/ B.
```

Check or_ind.

```
> and_ind  
: forall A B P:Prop, (A -> P) -> (B -> P) -> A \/ B -> P
```

- ▶ wprowadzenie alternatywy: taktyka constructor, apply or_introl lub left; apply or_intror lub right
- ▶ eliminacja alternatywy: taktyki destruct, elim, induction, case

KWANTYFIKATOR SZCZEGÓŁOWY

- ▶ formułę $\exists x : A. P(x)$ zapisujemy jako `exists x:A, P x`
- ▶ `exists x:A, P x` jest skrótem notacyjnym dla termu `ex P`
- ▶ `ex` jest zdefiniowany indukcyjnie (typ `A` jest ukryty)

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=  
  ex_intro : forall x : A, P x -> ex P
```

```
Check ex_ind.
```

```
> ex_ind : forall (A : Type) (P : A -> Prop) (P0 : Prop),  
  (forall x : A, P x -> P0) -> ex P -> P0
```

- ▶ dowód formuły egzystencjalnej składa się z termu-instancji kwantyfikatora oraz dowodu, że ten term spełnia własność `P`
- ▶ taktyka wprowadzenia – `constructor 1 with t, exists t, split with t` (musimy podać instancję dla `x`)
- ▶ taktyki eliminacji – `destruct, induction, elim, case`

WARIANTY TAKTYKI `constructor`

- ▶ `split` – gdy jest tylko jeden konstruktor typu (odp. `constructor 1`)
- ▶ `left`, `right` – gdy są dokładnie dwa konstruktory typu (odp. `constructor 1`, `constructor 2`)
- ▶ `exists t` – gdy jest 1 konstruktor i potrzebuje instancji zmiennych (odp. `constructor 1 with t`)

IZOMORFIZM CURRY'EGO-HOWARDA

- ▶ $\text{True} \Leftrightarrow \text{typ unit}$
- ▶ $\text{False} \Leftrightarrow \text{typ empty}$
- ▶ koniunkcja $\Leftrightarrow \text{typ produktu}$
- ▶ alternatywa $\Leftrightarrow \text{typ sumy}$

różnice: automatycznie generowana zasada indukcji, dopuszczalna eliminacja typu

KONIUNKCJA I TYP PRODUKTU

and_ind:

forall A B P : Prop, (A -> B -> P) -> A /\ B -> P

prod_ind:

forall (A B : Type) (P : A * B -> Prop),
 (forall (a : A) (b : B), P (a, b)) -> forall p : A * B, P p

ZASADA INDUKCJI DLA PREDYKATÓW

- ▶ dla predykatu P , Coq generuje automatycznie uproszczoną zasadę indukcji (termy-dowody nie są w niej uwzględniane) P_ind
- ▶ w razie potrzeby, możemy wygenerować pełną zasadę indukcji

`Scheme Tind := Induction for T Sort Prop.`

- ▶ i na odwrót: dla typów indukcyjnych (w `Set`, `Type`) możemy generować uproszczoną zasadę indukcji

`Scheme Tind := Minimality for T Sort Prop.`

ELIMINACJA DLA DEFINICJI INDUKCYJNYCH W Prop

Aby zachować zasadę *proof irrelevance*:

- ▶ nie można wykonać eliminacji (dopasowanie wzorca, indukcja) dla typu w Prop w celu otrzymania termu należącego do Set lub Type

```
Definition elimor (A B : Prop) (p : A \ / B) : bool :=  
  match p with  
  | or_introl _ => true  
  | or_intror _ => false  
end.
```

> Error: Incorrect elimination of "p" ...

- ▶ wyjątek: gdy predykat ma tylko jeden konstruktor, którego wszystkie argumenty są w Prop

```
Definition elimand (A B : Prop) (p : A /\ B) : bool :=  
  match p with  
  | conj _ _ => true  
end.
```


RÓWNOŚĆ

- ▶ operator logiczny definiowany indukcyjnie

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : x = x
```

- ▶ term `eq_refl x` dostarcza dowodu formuły `eq A x x`, czyli $x = x$ (dla odpowiedniego typu A)
- ▶ taktyki wprowadzenia równości: `apply eq_refl`, `constructor`, `reflexivity`

RÓWNOŚĆ

- ▶ zasada indukcji dla typu `eq` to reguła eliminacji równości (równość Leibniza)

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

- ▶ termy, które są równe w sensie relacji `eq`, są “wymienialne” w dowolnym kontekście
- ▶ można udowodnić
$$\text{forall } A \ B \ (f:A \rightarrow B) \ (x \ y:A), \ x = y \rightarrow f \ x = f \ y$$
- ▶ taktyki eliminacji równości: `rewrite`, `replace`, `induction`, ...

PREDYKATY INDUKCYJNE: PARZYSTOŚĆ

- ▶ `even` jest predykatem określonym na liczbach naturalnych
- ▶ `even n` zachodzi wtw `n` jest parzyste

```
Inductive even : nat -> Prop :=  
| even0 : even 0  
| evenSS : forall n:nat, even n -> even (S (S n)).
```

- ▶ `even0` jest dowodem formuły `even 0`
- ▶ `evenSS 0 even0` jest dowodem formuły `even 2`
- ▶ nie istnieje dowód `even (S 0)`

WNISKOWANIE O PREDYKATACH INDUKCYJNYCH

- ▶ indukcja po argumentach predykatów
- ▶ indukcja po definicji predykatu (trzeba uważać na argumenty ustalone)
- ▶ inwersja (taktyka *inversion*)

TAKTYKA inversion

- ▶ dana jest przesłanka $H : A \ t_1 \ \dots \ t_n$, gdzie A - predykat indukcyjny
- ▶ `inversion H` generuje wszystkie możliwe przypadki wyprowadzenia $A \ t_1 \ \dots \ t_n$ stosując konstruktory predykatu A
- ▶ każdy z tych przypadków generuje nowy cel do udowodnienia
- ▶ np. `even n` można otrzymać przez `even0` (wtedy $n = 0$) lub przez `evenSS m H'` (wtedy $n = S \ (S \ m)$) i $H' : \text{even } m$)

RELACJE JAKO PREDYKATY INDUKCYJNE

- ▶ predykaty indukcyjne mogą służyć do reprezentowania funkcji, np. pewnych funkcji rekurencyjnych niekoniecznie terminujących
- ▶ funkcję $f : A \rightarrow B$ możemy zdefiniować jako relację
- ▶ taką relację możemy reprezentować jako predykat indukcyjny $R_f : A \rightarrow B \rightarrow \text{Prop}$ (jeśli jej opis spełnia WPW)
- ▶ możemy wnioskować o tej funkcji, ale nie możemy wykonywać obliczeń z jej użyciem