Obliczenia i wnioskowanie w systemie Coq

Małgorzata Biernacka

Wykład 4
19.03.2013

Indukcyjne typy danych (przypomnienie)

```
Inductive bool : Set :=
| true : bool
| false : bool.

> bool_ind: ?
  bool_rec: ?
  bool_rect = ?
```

Indukcyjne typy danych (przypomnienie)

```
bool_ind =
fun P : bool -> Prop => bool_rect P
: forall P : bool -> Prop,
  P true -> P false -> forall b : bool, P b

bool_rect =
fun (P : bool -> Type)
(f : P true) (f0 : P false) (b : bool) =>
if b as b0 return (P b0) then f else f0
: forall P : bool -> Type,
  P true -> P false -> forall b : bool, P b
```

Indukcyjne typy danych (przypomnienie)

```
Parameter A : Set.

Inductive alist : Set :=
| nil : alist
| cons : A -> alist -> alist.

> alist_ind: ?
  alist_rec: ?
  alist_rect = ?
```

Indukcyjne typy danych (przypomnienie)

```
alist_rect =
fun (P : alist -> Type) (f : P nil)
  (f0 : forall (a : A) (a0 : alist), P a0 -> P (cons a a0)) =>
fix F (a : alist) : P a :=
  match a as a0 return (P a0) with
  | nil => f
  | cons a0 a1 => f0 a0 a1 (F a1)
  end
    : forall P : alist -> Type,
      P nil ->
      (forall (a : A) (a0 : alist), P a0 -> P (cons a a0)) ->
      forall a : alist, P a
```

Typy parametryczne: listy

- typ list – rodzina typów indeksowana typami z sortu `Set`

```
Inductive list (A:Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- każdy typ `list A` jest typem zdefiniowanym indukcyjnie
- `A` jest parametrem – jest widziany w środowisku w momencie deklarowania konstruktorów (jak w sekcji)
- szczególny przypadek typu zależnego, w którym każdy konstruktor musi mieć typ końcowy z tym samym parametrem
- zalety: prostsza zasada indukcji, możliwość ukrywania parametrów
- równoważny sposób definiowania typów parametrycznych – za pomocą sekcji (parametry jako zmienne deklarowane lokalnie)

Typy parametryczne: listy, c.d.

- `list : Set -> Set`
- pełne typy konstruktorów:

  `cons : forall A, A -> list A -> list A`

  `nil : forall A, list A`
- każdy konstruktor ma więc dodatkowy argument, który w zależności od kontekstu użycia może być pominięty (np. w konstrukcji `match` musi być pominięty)
- uwaga: typ biblioteczny `list` mieszka w `Type` (`list : Type -> Type`)

Typy parametryczne: listy

```
list_ind
     : forall (A : Type) (P : list A -> Prop),
       P nil ->
       (forall (a : A) (l : list A), P l -> P (a :: l)) ->
       forall l : list A, P l
```

Inne typy polimorficzne

- produkt i suma rozłączna

```
Inductive prod (A B : Type) : Type :=
| pair : A -> B -> prod A B.

Inductive sum (A B : Type) : Type :=
| inl : A -> sum A B
| inr : B -> sum A B.
```

- skróty notacyjne `*`, `+`

Argumenty niejawne (*implicit arguments*)

- dowolne argumenty/parametry, które Coq potrafi wywnioskować mogą być ustawione jako niejawne
- argumenty mogą być zadeklarowane jako niejawne *a priori* lub *a posteriori*
- *a priori*: w definicji, w nawiasach $\{\}$
- *a posteriori*: za pomocą komendy `Implicit Arguments id [A1,...,An].`
- jawne użycie argumentu niejawnego `(arg:=t)`
- deaktywacja ukrywania niejawnych argumentów `@id t ...`
- niejawne argumenty mogą być *(non) maximally inserted* (czyli term, którego argument jest niejawny, (nie) jest zaaplikowany do tego argumentu)
- automatyczna detekcja argumentów niejawnych przez komendę `Set Implicit Arguments.`

Typy i definicje wzajemnie rekurencyjne

- typy wzajemnie indukcyjne

```
Inductive tree : Set :=
| t_node : forest -> tree
with
forest : Set :=
| f_empty : forest
| f_cons : tree -> forest -> forest.
```

- funkcje wzajemnie rekurencyjne

```
Fixpoint tree_nodes (t:tree) : nat :=
match t with
| t_node f => forest_nodes f + 1
end
with
forest_nodes (f:forest) : nat :=
match f with
| f_empty => 0
| f_cons t f => tree_nodes t + forest_nodes f
end.
```

Indukcyjne typy zagnieżdżone

- definiowany typ może występować wewnątrz typu argumentu konstruktora

```
Inductive foo : Type :=
| c0 : nat -> foo
| c1 : list foo -> foo.
```

- funkcje rekurencyjne operujące na typie zagnieżdżonym odzwierciedlaja jego strukturę

```
Fixpoint fmap (f: nat -> nat) (t:foo) : foo :=
match t with
| c0 n => c0 (f n)
| c1 l => c1 (map (fmap f) l)
end.
```

Problemy z indukcją

- często zasada indukcji generowana automatycznie dla typu indukcyjnego jest za słaba
- Coq nie uwzględnia wzajemnej indukcji ani zagnieżdżenia typów
- co można zrobić:
  - wygenerować mocniejszą zasadę indukcji za pomocą komendy `Scheme` (dla typów wzajemnie indukcyjnych)
  - zdefiniować własną zasadę indukcji (dla typów zagnieżdżonych)

Indukcyjne typy zagnieżdżone

- automatyczna zasada indukcji:

```
tree_ind
: forall P : tree -> Prop,
  (forall f : forest, P (t_node f)) -> forall t : tree, P t
```

- specjalna zasada indukcji

```
Scheme t_ind := Induction for tree Sort Prop
with f_ind := Induction for forest Sort Prop.

Check t_ind.

t_ind
     : forall (P : tree -> Prop) (P0 : forest -> Prop),
       (forall f : forest, P0 f -> P (t_node f)) ->
       P0 f_empty ->
       (forall t : tree, P t ->
          forall f1 : forest, P0 f1 -> P0 (f_cons t f1)) ->
          forall t : tree, P t
```

Ręcznie robiona zasada indukcji

- dla typów wzajemnie indukcyjnych wymaga funkcji wzajemnie rekurencyjnych
- dla typów zagnieżdżonych wymaga zagnieżdżonej rekurencji (definicje wzajemnie rekurencyjne nie działają, bo nie spełniają warunku terminacji)

Jeśli dowód przez indukcję nie przechodzi

- może trzeba użyć mocniejszej zasady indukcji
- może trzeba wzmocnić twierdzenie, którego dowodzimy
- trzeba uważać na kolejność argumentów w twierdzeniach dowodzonych indukcyjnie – argumenty stojące przed indukcyjnym zostaną ustalone w hipotezie ind. (być może trzeba je przepermutować)
- taktyka `case` "gubi" informacje na temat struktury termu (wtedy trzeba używać `case_eq` lub `generalize`)
- taktyki `elim, case` nie modyfikują kontekstu

Silniej specyfikowane listy

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

Check cons 43 (le_n 43) nil.
```

Termy typu `mlist` zawierają podtermy, które są dowodami.

Silniej specyfikowane listy

Możemy udowodnić, że wszystkie elementy listy typu mlist są ¿ 42.

```
Fixpoint all_elem_gt_42 (l:mlist) : Prop :=
match l with
| nil => True
| cons a _ l' => (a > 42) /\ all_elem_gt_42 l'
end.

Theorem all_gt_42 : forall l : mlist, all_elem_gt_42 l.
```