Obliczenia i wnioskowanie w systemie Coq

Małgorzata Biernacka

Wykład 3
07.03.2019

Definicje indukcyjne a Coq

- indukcyjne typy danych/predykaty są definiowalne w CoC, ale taki sposób reprezentacji ma poważne wady
- CIC jest rozszerzeniem CoC, w którym definicje indukcyjne są pojęciem pierwotnym

  Ch. Paulin-Mohring "Inductive Definitions in the System Coq. Rules and Properties" 1992

Definicje indukcyjne typów

- elementy typu indukcyjnego są definiowane za pomocą konstruktorów (odpowiadających regułom indukcyjnym)
- element typu indukcyjnego można otrzymać wyłącznie za pomocą konstruktorów
- konstruktory są parami różne (termy-wyprowadzenia z nich zbudowane są różne)
- konstruktory są różnowartościowe (tj. jeśli `c t = c s`, to `t = s`)
- użycie: konstruowanie termów, eliminacja termów, dowodzenie własności przez indukcję

Definicje indukcyjne typów

- zbiór termów reprezentujących liczby naturalne możemy zapisać w składni Ocamla tak:

```
type nat = O | S of nat
```

- w Coqu:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

- wygodnie jest myśleć o definicji indukcyjnej używając reguł wyprowadzenia/wnioskowania:

$$\frac{}{\vdash O : nat} \qquad \frac{\vdash t : nat}{\vdash S t : nat}$$

Definicje indukcyjne typów w Coqu

Definicja użytkownika:

- nazwa typu jest stałą (termem) o typie końcowym `Set` lub `Type` (`nat : Set`, `bool : Set`, `list : Type -> Type`, `array : nat -> Set`)
- każdy konstruktor typu T jest stałą (termem) o typie końcowym T

Warunek pozytywnych wystąpień (*strict positivity condition*) w definicjach indukcyjnych

- typ T spełnia warunek pozytywnych wystąpień dla identyfikatora X w termie T :
  - T postaci $XM_1 \dots M_n$: X nie występuje w M_i
  - T postaci $\forall x : T'.U$: X występuje ściśle pozytywnie w T' oraz U spełnia war. poz. wyst. dla X
- X występuje ściśle pozytywnie w T, gdy:
  - X nie występuje w T
  - $T = XM_1 \dots M_n$ i X nie występuje w M_i
  - $T = \forall x : T'.U$ i X nie występuje w T', występuje ściśle pozytywnie w U
  - w pewnych przypadkach w parametrach typu indukcyjnego T (dokładniej: patrz Reference Manual)

Warunek pozytywnych wystąpień (*strict positivity condition*) w definicjach indukcyjnych

Negatywne wystąpienia identyfikatora typu definiowanego indukcyjnie są zabronione w typach argumentów konstruktorów tego typu

```
Inductive A : Set :=
| c : (A -> B) -> A.
```

(Error: Non strictly positive occurrence of "`A`" in "`(A->B)->A`")

Po co warunek pozytywnych wystąpień?

W przeciwnym razie

- nie zachodziłaby własność silnej normalizacji termów
- konwersja i sprawdzanie typów nie byłyby rozstrzygalne
- system byłby logicznie sprzeczny

```
Inductive A : Prop/Set :=
| c : (A -> B) -> A.

Definition F (t:A) : B :=
match t with
| c f => f t
end.

F (c F) -> ...
```

Definicje indukcyjne typów w Coqu

Każda definicja indukcyjna typu `T` powoduje dodanie do środowiska:

- definicji stałej `T` odpowiedniego typu
- deklaracji konstruktorów typu `T` jako stałych odpowiedniego typu (postaci `c : forall ..., T`)
- definicji stałych `T_ind`, `T_rect`, `T_rec` wygenerowanych automatycznie na podstawie definicji typu `T`
- `T_ind` – schemat indukcji strukturalnej dla `T`
- `T_rec` – schemat rekursji prostej dla `T`

Dopasowanie wzorca

- składnia `match`

```
Definition iszero (n:nat) : Prop :=
match n with
| O => True
| S m => False
end
```

- definicja za pomocą `match` musi obejmować wszystkie przypadki

Definicje rekurencyjne

- za pomocą komendy `Fixpoint`
- lub za pomocą wyrażenia `fix` (gdy definiujemy funkcję rekurencyjną lokalnie)

```
Fixpoint fact (n:nat) {struct n} : nat :=
match n with
| O => 1
| S m => n * fact m
end.

Definition fact2 := fix fact (n:nat) : nat :=
match n with
| O => 1
| S m => n * fact m
end.
```

- syntaktyczne kryterium terminacji – wywołanie rekurencyjne musi dotyczyć właściwego podtermu wskazanego argumentu

ι-redukcja

- `match` + konstruktory typu wprowadzają nowy rodzaj redeksu
- ι-redukcja dla typu T o konstruktorach c_i (każdy o k_i argumentach), $1 \leq i \leq n$:

$$
\begin{array}{l}
\texttt{match}\ c_i\ a_{;}\ \texttt{...}\ a_{k_i}\ \texttt{with} \\
\texttt{|}\ c_1\ p_1\ \texttt{...}\ p_{k_1}\ \texttt{=>}\ t_1 \\
\texttt{|}\ \texttt{...} \\
\texttt{|}\ c_n\ p_1\ \texttt{...}\ p_{k_n}\ \texttt{=>}\ t_n \\
\texttt{end} \\
\to t_i[a_1/p_1; a_2/p_2; \ldots; a_{k_i}/p_{k_i}]
\end{array}
$$

Zasada indukcji

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n))
       -> forall n : nat, P n
```

```
nat_rect =
fun (P : nat -> Type) (f : P 0)
                      (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | S n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

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 `*`, `+`

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)

Taktyki dla eliminacji typów indukcyjnych

- `induction n`
- `elim` t
- `case` t
- `destruct H`
- `discriminate H`
- `injection H`

Taktyka `induction`

- `induction t` – wykonuje indukcję ze względu na typ termu `t` (musi być indukcyjny):
  - bieżący cel jest traktowany jako parametr `P` w definicji zasady indukcji
  - taktyka generuje nowe podcele dla każdego z konstruktorów typu
  - taktyka wprowadza do przesłanek odpowiednie hipotezy indukcyjne

Taktyka `elim`

- `elim t` – prostsza niż `induction`
  - wyszukuje odpowiednią zasadę eliminacji dla typu `t`
  - generuje nowe podcele dla każdego z konstruktorów typu
  - nie modyfikuje kontekstu
  - działa jak `apply ...`

Taktyka `case`

- `case t` – dowód przez rozpatrywanie przypadków; typ termu `t` musi być indukcyjny
- działa podobnie do `elim t`

Taktyka `destruct`

- `destruct t` – dowód przez rozpatrywanie przypadków; typ termu `t` musi być indukcyjny
- działa podobnie do `induction t`, ale nie generuje hipotezy indukcyjnej

Taktyka `discriminate`

- taktyka wykorzystuje fakt, że dwa termy zbudowane za pomocą dwóch różnych konstruktorów tego samego typu nie mogą być równe
- `discriminate H` pozwala udowodnić dowolny cel, gdy w przesłance `H : t1 = t2` postaci normalne termów `t1` i `t2` mają w pewnym podtermie różne konstruktory na tym samym miejscu
- działanie `discriminate` można zasymulować za pomocą prostszych taktyk

Taktyka `injection`

- taktyka wykorzystuje fakt, że konstruktory typu są różnowartościowe
- jeśli `c t`$_1$ `...  t`$_n$ `= c s`$_1$ `...  s`$_n$, to `t`$_i$ `= s`$_i$ dla $i = 1, \ldots, n$
- `injection H` generuje cel `t`$_1$ `= s`$_1$ `-> ...  -> t`$_n$ `= s`$_n$ `-> G`, gdy `H : c t`$_1$ `...  t`$_n$ `= c s`$_1$ `...  s`$_n$
- działanie `injection` można zasymulować za pomocą prostszych taktyk

Taktyki wprowadzania typów indukcyjnych

- `constructor` – zastosowanie kontruktora do zbudowania termu typu indukcyjnego
- `apply c` – j.w. (c jest konstruktorem)