Typy indukcyjne w Coqu

Definicja

Oto definicja typu indukcyjnego reprezentującego polimorficzne listy z długością, tj. dla dowolnego A typu `Set` i dowolnego n typu `nat`, `(listn` A n`)` oznacza typ "lista elementów typu A, długości n"

```
Inductive listn [A:Set] : nat -> Set :=
| niln : (listn A O)
| consn : A -> (n:nat)(listn A n) -> (listn A (S n)).
```

Po wprowadzeniu tej definicji typy `listn` i jego konstruktorów są następujące:

```
listn : Set->nat->Set
niln  : (A:Set)(listn A O)
consn : (A:Set)A->(n:nat)(listn A n)->(listn A (S n))
```

Na definicję indukcyjną skłądają się następujace elementy:

- nazwa typu indukcyjnego – `listn`,
- lista parametrów – `[A:Set]`

  Parametry to niezmienne argumenty typu indukcyjnego. Wszystkie wystąpienia `listn` w jego definicji muszą być zaaplikowane go parametru `A`. Z tego powodu kolejny argument `listn` typu `nat` nie mógłby być parametrem. Oprócz tego typy konstruktorów oraz samego `listn` są abstrahowane ze względu na parametry.
- typ (rodzaj) typu indukcyjnego – `nat -> Set`

  Musi to być produkt (może być zależny) zakończony jednym z sortów (`Set`, `Prop` lub `Type`). Jeśli sortem tym jest `Set` mówimy o *typie* indukcyjnym, jeśli `Prop` – o *predykacie* 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, czyli w tym przypadku `(listn A _)` gdzie ostatnia pozycja zależy od konstruktora.

Dodatkowym warunkiem, który muszą spełniać typy konstruktorów jest warunek pozytywności. Mówi on, że `listn` może wystąpić w typie *argumentu* konstruktora wyłącznie na końcu, tzn. argumenty konstruktora mogą być postaci *coś*$_1$ `-> ... ->` *coś*$_n$ `-> (listn A _)`, gdzie w *coś*$_i$ nie występuje `listn`. Następująca deklaracja konstruktora:

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

nie jest poprawna.

Case – analiza przypadków

Załóżmy, 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:(n:nat)(listn A n) -> Prop;
d_pusta : (P O (niln A));
d_niepusta : (n':nat)(a':A)(l':(list A n')) -> (P (S n') (consn A a' n' l'));
n:nat; l:nat
==============================================================
<P>Cases l of
     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. Mogłaby to być też jakaś konkretna stała, np. typ `nat` lub `bool`
- najogólniejszy typ `P` to "argumenty `listn -> (listn` z tymi argumentami `) ->` sort"

  Predykat `P` mógłby mieć też prostszy typ "argumenty `listn ->` sort", czyli `nat ->` sort, jak `[n:nat](listn B n)` (dla jakiegoś `B:Set`) lub `[n:nat](Even n)` albo być stałą typu "sort", jak `nat`, czy `bool`. W tym ostatnim przypadku nie jest konieczne umieszczanie <P> przes konstrukcją `Cases`.

  Możliwym skomplikowanym predykatem `P`, jest np.
  `[n:nat;l:(list A n)](niepusta A (S n) (consn A a n l))`
  (gdzie `niepusta` to hipotetyczny zdefiniowany wcześniej predykat, typu
  `(A:Set)(n:nat)(listn A n) -> Prop`, a `a` to jakiś element typu `A`).
- typy gałęzi instrukcji `Cases` muszą odpowiadać typom konstruktorów, tylko na końcu zamiast `(listn` argumenty`)` stoi `P` zaaplikowane do odpowiedniego argumentu i do listy skonstruowanej danym konstruktorem.
- zauważmy, że po lewej stronie `=>` konstruktory występują bez parametru `A`. Po prawej muszą już mieć parametr.

Z instrukcją `Cases` związana jest następująca reguła, zwana ι (jota) redukcją:

```
Cases (niln A) of niln => d_puste | ... end   ⟶ι   d_puste
Cases (consn a' n' l') of ... | (consn a n l) => (d_niepusta n a l) end
                                                ⟶ι   (d_niepuste n' a' l')
```

Rekurencja strukturalna

Rekurencja strukturalna służy zarówno do definiowania funkcji jak i do robienia dowodów indukcyjnych. W Coqu istnieje konstrukcja `Fix` do definiowania funkcji lokalnych oraz jego uproszczona wersja, używająca słowa kluczowego `Fixpoint`, służąca do definiowania funkcji globalnych.

Najczęściej używana składnia operatora `Fix`, to:

$$\begin{aligned}\texttt{Fix}\ \ f_i\{ & f_1\ [x_1^1 : t_1^1; \ldots ; x_{i_1}^1 : t_{i_1}^1] : t_{k_1}^1 := d_1\ \ \texttt{with}\\ & \ldots\ \ \texttt{with}\\ & f_n\ [x_1^n : t_1^n; \ldots ; x_{i_n}^n : t_{i_n}^n] : t_{k_n}^n := d_n\}\end{aligned}$$

Funkcje $f_1 \ldots f_n$ 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 i_j. Typ $t_{i_j}^j$ musi zatem być typem indukcyjnym.

W ciałach funkcji ($d_1 \ldots d_n$) może występować dowolne f_j, ale musi być zaaplikowane do co najmniej i_j, argumentów, z czego ostatni musi być *strukturalnie mniejszy* niż argument wywołania funkji, czyli "strzeżony" poprzez co najmniej jedną konstrukcję `Cases`.

Powyższa składnia operatora `Fix` jest równoważna następującej:

$$\begin{aligned}\texttt{Fix}\ \ f_i\{ & f_1/i_1 : (x_1^1 : t_1^1; \ldots ; x_{i_1}^1 : t_{i_1}^1)t_{k_1}^1 := [x_1^1 : t_1^1; \ldots ; x_{i_1}^1 : t_{i_1}^1]d_1\ \ \texttt{with}\\ & \ldots\ \ \texttt{with}\\ & f_n/i_n : (x_1^n : t_1^n; \ldots ; x_{i_n}^n : t_{i_n}^n)t_{k_n}^n := [x_1^n : t_1^n; \ldots ; x_{i_n}^n : t_{i_n}^n]d_n\}\end{aligned}$$

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

Reguła typowania jest zgodna z oczekiwaniem:

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

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

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

$$\texttt{Fix}\ \ f_i\{\Gamma_F\}\ a_1 \ldots a_{i_i} \longrightarrow_\iota D_i\{f_j \mapsto \texttt{Fix}\ \ f_j\{\Gamma_F\}\}\ a_1 \ldots a_{i_i}$$

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

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

$$\begin{aligned}\Gamma_F \equiv & f_1\ [x_1^1 : t_1^1; \ldots ; x_{i_1}^1 : t_{i_1}^1] : t_{k_1}^1 := d_1\ \ \texttt{with}\\ & \ldots\ \ \texttt{with}\\ & f_n\ [x_1^n : t_1^n; \ldots ; x_{i_n}^n : t_{i_n}^n] : t_{k_n}^n := d_n\end{aligned}$$

Wprowadzenie komendy:

`Fixpoint` Γ_F`.`

odpowiada wprowadzeniu następującego ciągu komend:

`Definition` f_1`:=Fix` $f_1\{\Gamma_F\}$`.`
`...`
`Definition` f_n`:=Fix` $f_n\{\Gamma_F\}$`.`

Kilka przykładów

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

```
Fixpoint npol_podloga [n:nat] : nat :=
  Cases n of
    O => O
  | (S n') => (npol_sufit n')
  end
with npol_sufit [n:nat] : nat :=
  Cases n of
    O => O
  | (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 wejsciowej, pierwsza z nich zaczyna od pierwszego elementu, a druga – od drugiego. Ich typy (a konkretnie dlugości list wyjściowych) opisane są za pomocą `npol_podloga` i `npol_sufit`.

```
Fixpoint codrugi_zpierwszym [A:Set;n:nat; l:(listn A n)]
          : (listn A (npol_sufit n)) :=
  <[n](listn A (npol_sufit n))>Cases l of
    niln => (niln A)
  | (consn a n' l') => (consn A a ? (codrugi_bezpierw A n' l'))
  end
with codrugi_bezpierw [A:Set;n:nat; l:(listn A n)]
          : (listn A (npol_podloga n)) :=
  <[n](listn A (npol_podloga n))>Cases l of
    niln => (niln A)
  | (consn a n' l') => (codrugi_zpierwszym A n' l')
  end.
```

Obiekty indukcyjne i dowodzenie

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 sytuacje:

```
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 `Cases`. Służy do tego taktyka `Case`. Tworzy ona tyle "pod-celów" ile jest konstruktorów danego typu. Uwaga! Jest ona dosyć głupia – dla typów indukcyjnych zależnych bierze również pod uwagę przypadki "niemożliwe" (np. lista o długości Øzaczynająca się od `consn`)

Różne warianty `Case` (robiące automatycznie `Intros` i/lub czyszczenie niepotrzebnych hipotez), to `Destruct` oraz `NewDestruct`.

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

W zasadzie `Elim n.` odpowiada `Apply nat_ind.` o ile `nat` jest typem `n` oraz "cel" jest w sorcie `Prop`.

Wariantami `Elim` są `Induction` i `NewInduction`.

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 czegokolwiek.

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

Ich sprytnym połączeniem jest taktyka `Simplify_eq`.

Taktyką, która stara się nadrobić niedostatki taktyki `Case` 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`.

Ponieważ lemat generowany przez `Inversion` jest spory, w przypadku, gdy wielokrotnie używamy tej taktyki dla tej samej konstrukcji indukcyjnej, możemy nadać temu lematowi nazwę, a następnie używać go poprzez `Apply`. Do generowania służy polecenie `Derive Inversion`.