Programowanie z typami zależnymi i dowodzenie twierdzeń

Rekurencja

26.05.2015

Rekurencja strukturalna—przykład div2

```
Fixpoint div2 n :=
 match n with
 | S (S n') => S (div2 n')
 | _=> 0
 end.
```

Przykład div2'

```
Definition pred n :=
 match n with  | O => n  | S n' => n'  end.
```

```
Fixpoint div2' n :=
 match n with
 | S n' => S (div2' (pred n'))
 | _=> 0
 end.
```

Przykład div2'

```
Definition pred n :=
 match n with  | O => n  | S n' => n'  end.
```

```
Fixpoint div2' n :=
 match n with
 | S n' => S (div2' (pred n'))
 | _=> O
 end.
```

Liczby całkowite

```
Inductive positive : Set :=
 | xI : positive -> positive
 | xO : positive -> positive
 | xH : positive.
```

```
Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive -> Z
 | Zneg : positive -> Z.
```

- xH to 1
- xO to kod $2 * p$
- xI to kod $2 * p + 1$
- wydajniejsze od nat

Liczby całkowite

```
Inductive positive : Set :=
 | xI : positive -> positive
 | xO : positive -> positive
 | xH : positive.
```

```
Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive -> Z
 | Zneg : positive -> Z.
```

- xH to 1
- xO to kod $2 * p$
- xI to kod $2 * p + 1$
- wydajniejsze od nat

Liczby całkowite

```
Inductive positive : Set :=
 | xI : positive -> positive
 | xO : positive -> positive
 | xH : positive.
```

```
Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive -> Z
 | Zneg : positive -> Z.
```

- xH to 1
- xO to kod $2 * p$
- xI to kod $2 * p + 1$
- wydajniejsze od nat

Liczby całkowite

```
Inductive positive : Set :=
 | xI : positive -> positive
 | xO : positive -> positive
 | xH : positive.
```

```
Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive -> Z
 | Zneg : positive -> Z.
```

- xH to 1
- xO to kod $2 * p$
- xI to kod $2 * p + 1$
- wydajniejsze od nat

Liczby całkowite

```
Inductive positive : Set :=
 | xI : positive -> positive
 | xO : positive -> positive
 | xH : positive.
```

```
Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive -> Z
 | Zneg : positive -> Z.
```

- xH to 1
- xO to kod $2 * p$
- xI to kod $2 * p + 1$
- wydajniejsze od nat

Liczby całkowite

```
Inductive positive : Set :=
 | xI : positive -> positive
 | xO : positive -> positive
 | xH : positive.
```

```
Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive -> Z
 | Zneg : positive -> Z.
```

- xH to 1
- xO to kod $2 * p$
- xI to kod $2 * p + 1$
- wydajniejsze od nat

Rekursja w `factZ` nie jest strukturalna

```
Fixpoint factZ (x : Z): Z :=
  if Zle_bool x 0 then 1 else x * factZ (x - 1).
```

$x - 1$ nie jest strukturalnie mniejsze od x

Predykaty Acc i well-founded

```
Print Acc.
```

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

```
well_founded =
fun (A : Type) (R : A → A → Prop) ⇒ ∀ a : A, Acc R a
```

Przykłady relacji dobrze ufundowanych to: `lt` na typie `nat`, `Zwf c` na `Z` (czyli $<$ na przedziale $[c, +\infty)$)

Predykaty Acc i well-founded

```
Print Acc.
```

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

```
well_founded =
fun (A : Type) (R : A → A → Prop) ⇒ ∀ a : A, Acc R a
```

Przykłady relacji dobrze ufundowanych to: `lt` na typie `nat`, `Zwf c` na `Z` (czyli $<$ na przedziale $[c, +\infty)$)

Predykaty Acc i well-founded

```
Print Acc.
```

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

```
well_founded =
fun (A : Type) (R : A → A → Prop) ⇒ ∀ a : A, Acc R a
```

Przykłady relacji dobrze ufundowanych to: `lt` na typie `nat`, `Zwf c` na `Z` (czyli $<$ na przedziale $[c, +\infty)$)

Inna definicja dobrego ufundowania

```
CoInductive infiniteDecreasingChain A (R : A → A → Prop) :
stream A → Prop :=
  | ChainCons : ∀ x y s, infiniteDecreasingChain R (Cons y s)
    → R y x
    → infiniteDecreasingChain R (Cons x (Cons y s)).
```

Związki między definicjami

```
Lemma noBadChains' : ∀ A (R : A → A → Prop) x, Acc R x
    → ∀ s, ¬infiniteDecreasingChain R (Cons x s).
    induction 1; crush;
      match goal with
        | [ H : infiniteDecreasingChain _ _ ⊢ _ ] ⇒ inversion H;
eauto
      end.
  Qed.

Theorem noBadChains : ∀ A (R : A → A → Prop), well_founded R
    → ∀ s, ¬infiniteDecreasingChain R s.
    destruct s; apply noBadChains'; auto.
  Qed.
```

W drugą stronę prawdziwe tylko w logice klasycznej

Związki między definicjami

```
Lemma noBadChains' : ∀ A (R : A → A → Prop) x, Acc R x
    → ∀ s, ¬infiniteDecreasingChain R (Cons x s).
    induction 1; crush;
      match goal with
        | [ H : infiniteDecreasingChain _ _ ⊢ _ ] ⇒ inversion H;
eauto
      end.
  Qed.

Theorem noBadChains : ∀ A (R : A → A → Prop), well_founded R
    → ∀ s, ¬infiniteDecreasingChain R s.
    destruct s; apply noBadChains'; auto.
  Qed.
```

W drugą stronę prawdziwe tylko w logice klasycznej

Związki między definicjami

```
Lemma noBadChains' : ∀ A (R : A → A → Prop) x, Acc R x
    → ∀ s, ¬infiniteDecreasingChain R (Cons x s).
    induction 1; crush;
      match goal with
        | [ H : infiniteDecreasingChain _ _ ⊢ _ ] ⇒ inversion H;
eauto
      end.
  Qed.

Theorem noBadChains : ∀ A (R : A → A → Prop), well_founded R
    → ∀ s, ¬infiniteDecreasingChain R s.
    destruct s; apply noBadChains'; auto.
  Qed.
```

W drugą stronę prawdziwe tylko w logice klasycznej

well_founded_induction i Fix

```
Check well_founded_induction.

well_founded_induction
     : ∀ (A : Type) (R : A → A → Prop),
       well_founded R →
       ∀ P : A → Type,
       (∀ x : A, (∀ y : A, R y x → P y) → P x) →
       ∀ x : A, P x
```

- Fix ma taki sam typ
- well_founded R zapewnia, że nie ma nieskończonego ciągu wywołań rekurencyjnych

well_founded_induction i Fix

```
Check well_founded_induction.

well_founded_induction
     : ∀ (A : Type) (R : A → A → Prop),
       well_founded R →
       ∀ P : A → Type,
       (∀ x : A, (∀ y : A, R y x → P y) → P x) →
       ∀ x : A, P x
```

- Fix ma taki sam typ
- well_founded R zapewnia, że nie ma nieskończonego ciągu wywołań rekurencyjnych

well_founded_induction i Fix

```
Check well_founded_induction.

well_founded_induction
     : ∀ (A : Type) (R : A → A → Prop),
       well_founded R →
       ∀ P : A → Type,
       (∀ x : A, (∀ y : A, R y x → P y) → P x) →
       ∀ x : A, P x
```

- `Fix` ma taki sam typ
- `well_founded R` zapewnia, że nie ma nieskończonego ciągu wywołań rekurencyjnych

Dzielenie całkowite — wfdiv

```
Definition wfdiv :
 forall m n:nat, 0 < n -> {q:nat &{r:nat | m = q*n+r ∧ r < n}}
:=
 well_founded_induction lt_wf div_type div_F.
```

div_F definiowane w pliku wfdiv.v

Dzielenie całkowite — wfdiv

```
Definition wfdiv :
 forall m n:nat, 0 < n -> {q:nat &{r:nat | m = q*n+r ∧ r < n}}
:=
 well_founded_induction lt_wf div_type div_F.
```

div_F definiowane w pliku wfdiv.v

Dzielenie całkowite — div — używając Function

```
Function div (p:nat*nat) {measure fst} : nat*nat :=
match p with
 | (_,0) => (0,0)
 | (a,b) => if le_lt_dec b a
    then let (x,y):=div (a-b,b) in (1+x,y)
    else (0,a)
end.

intros.
simpl.
omega.
Defined.
```

Komenda Function — przykład factZ

```
Require Import Recdef.

Function factZ (x : Z) {measure Zabs_nat x} : Z :=
  if Zle_bool x 0 then 1 else x * factZ (x - 1).
```

pozostaje zrobić dowód, że miara maleje...(w pliku gr.v)

Dowodzenie własności

```
Lemma div2_le : forall n, div2 n <= n.
Proof.
 induction n.
 auto.
 simpl.
 destruct n.
 auto.
 (* Does not work: bad induction hypothesis *)
Abort.
```

Lepiej najpierw dowieść: `(forall m n, n <= m -> div2 n <= n).`

Functional induction

```
Functional Scheme div2_ind := Induction for div2 Sort Prop.
Check div2_ind.

div2_ind
  : forall P : nat -> nat -> Prop,
  (forall n : nat, n = 0 -> P 0 0) ->
  (forall n n0 : nat, n = S n0 -> n0 = 0 -> P 1 0) ->
  (forall n n0 : nat,
  n = S n0 ->
  forall n' : nat,
  n0 = S n' -> P n' (div2 n') -> P (S (S n')) (S (div2 n')))
->
  forall n : nat, P n (div2 n)
```

Functional induction

```
Check div2_equation.

div2_equation
  : forall n : nat,
  div2 n = match n with
     | 0 => 0
     | 1 => 0
     | S (S n') => S (div2 n')
     end
```

Dowód div2_le''

```
Lemma div2_le'' : forall x, div2 x * 2 <= x.
Proof.
intros x; functional induction div2 x; auto with zarith.
Qed.
```

functional induction generowane automatycznie gdy używamy `Function`

Powrót do div — poprawność dzielenia

```
Function generuje div_ind

Definition divPre (args:nat*nat) : Prop := (snd args)<>0.

Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
 let (n,d):=args in let (q,r):=res in q*d+r=n / r<d.

Theorem div_correct : forall (p:nat*nat), divPre p -> divRel
p (div p).
```

Dowód w pliku div.v

mergeSort używający well_founded

```
Section mergeSort.
  Variable A : Type.
  Variable le : A → A → bool.

  Fixpoint insert (x : A) (ls : list A) : list A :=
    match ls with
      | nil ⇒ x :: nil
      | h :: ls' ⇒
        if le x h
          then x :: ls
          else h :: insert x ls'
    end.
```

mergeSort używający well_founded

```
Section mergeSort.
  Variable A : Type.
  Variable le : A → A → bool.

  Fixpoint insert (x : A) (ls : list A) : list A :=
    match ls with
      | nil ⇒ x :: nil
      | h :: ls' ⇒
        if le x h
          then x :: ls
          else h :: insert x ls'
    end.
```

mergeSort używający well_founded

```
Fixpoint merge (ls1 ls2 : list A) : list A :=
  match ls1 with
    | nil ⇒ ls2
    | h :: ls' ⇒ insert h (merge ls' ls2)
  end.
```

```
Fixpoint split (ls : list A) : list A × list A :=
  match ls with
    | nil ⇒ (nil, nil)
    | h :: nil ⇒ (h :: nil, nil)
    | h1 :: h2 :: ls' ⇒
      let (ls1, ls2) := split ls' in
        (h1 :: ls1, h2 :: ls2)
  end.
```

mergeSort używający well_founded

```
Fixpoint merge (ls1 ls2 : list A) : list A :=
  match ls1 with
    | nil ⇒ ls2
    | h :: ls' ⇒ insert h (merge ls' ls2)
  end.
```

```
Fixpoint split (ls : list A) : list A × list A :=
  match ls with
    | nil ⇒ (nil, nil)
    | h :: nil ⇒ (h :: nil, nil)
    | h1 :: h2 :: ls' ⇒
      let (ls1, ls2) := split ls' in
        (h1 :: ls1, h2 :: ls2)
  end.
```

Zła definicja

```
Fixpoint mergeSort (ls : list A) : list A :=
    if leb (length ls) 1
      then ls
      else let lss := split ls in
        merge (mergeSort (fst lss)) (mergeSort (snd lss)).

Recursive call to mergeSort has principal argument equal to
" fst (split ls)" instead of a subterm of " ls".
```

Relacja porządku `lengthOrder`

```
Definition lengthOrder (ls1 ls2 : list A) :=
    length ls1 < length ls2.
```

```
Lemma lengthOrder_wf' : ∀ len, ∀ ls, length ls ≤ len → Acc lengthOrder ls.
```

```
Theorem lengthOrder_wf : well_founded lengthOrder.
```

Relacja porządku `lengthOrder`

```
Definition lengthOrder (ls1 ls2 : list A) :=
    length ls1 < length ls2.

Lemma lengthOrder_wf' : ∀ len, ∀ ls, length ls ≤ len → Acc lengthOrder
ls.

Theorem lengthOrder_wf : well_founded lengthOrder.
```

Relacja porządku `lengthOrder`

```
Definition lengthOrder (ls1 ls2 : list A) :=
    length ls1 < length ls2.

Lemma lengthOrder_wf' : ∀ len, ∀ ls, length ls ≤ len → Acc lengthOrder
ls.

Theorem lengthOrder_wf : well_founded lengthOrder.
```

Definicja mergeSort używająca Fix

```
Definition mergeSort : list A → list A.
  refine (Fix lengthOrder_wf (fun _ ⇒ list A)
    (fun (ls : list A)
      (mergeSort : ∀ ls' : list A, lengthOrder ls' ls → list A) ⇒
      if le_lt_dec 2 (length ls)
        then let lss := split ls in
          merge (mergeSort (fst lss) _) (mergeSort (snd lss) _)
        else ls)); subst lss; eauto.
  Defined.
End mergeSort.
```

Definicja mergeSort' używająca Function

```
Function mergeSort' (ls : list A) {measure length ls} :
list A :=
 if le_lt_dec 2 (length ls)
 then let lss := split ls in
   merge (mergeSort' (fst lss)) (mergeSort' (snd lss))
 else ls.
intros.
...
Defined.
```

Lepsze Merge

```
Definition slen (p:list nat * list nat) :=
 length (fst p) + length (snd p).

Function Merge (p:list nat * list nat) { measure slen p } :
list nat :=
 match p with
 | (nil, l2) => l2
 | (l1, nil) => l1
 | ((x1::l1') as l1, (x2::l2') as l2) =>
  if leb x1 x2 then x1::Merge (l1',l2)
  else x2::Merge (l1,l2')
 end.

(* first goal *) unfold slen;simpl;intros. omega.
(* second goal *) unfold slen;simpl;intros. omega.
Defined.
```

Dowody przez functional induction

Indukcja najlepiej po zmiennej. Patrz plik gr.v

```
Lemma Merge_count : forall x l1 l2,
 count x (Merge (l1, l2)) = count x l1 + count x l2.
Proof.
(* intros x l1 l2;functional induction (Merge (l1,l2)). *)
(* Some information is lost, we need generalization *)
...
```

```
Lemma Merge_count' : forall x l1 l2,
 count x (Merge (l1, l2)) = count x l1 + count x l2.
Proof.
intros x l1 l2.
remember (l1,l2) as p.
revert l1 l2 Heqp.
functional induction (Merge p)
...
```

Dowody przez functional induction

Indukcja najlepiej po zmiennej. Patrz plik gr.v

```
Lemma Merge_count : forall x l1 l2,
 count x (Merge (l1, l2)) = count x l1 + count x l2.
Proof.
(* intros x l1 l2;functional induction (Merge (l1,l2)). *)
(* Some information is lost, we need generalization *)
...
```

```
Lemma Merge_count' : forall x l1 l2,
 count x (Merge (l1, l2)) = count x l1 + count x l2.
Proof.
intros x l1 l2.
remember (l1,l2) as p.
revert l1 l2 Heqp.
functional induction (Merge p)
...
```