Programowanie z typami zależnymi i dowodzenie twierdzeń

Typy koindukcyjne

19.05.2015

Typ leniwych list — LList

```
CoInductive LList (A:Set) :Set :=
   LNil : LList A
|  LCons : A -> LList A -> LList A
```

- termy zbudowane tylko z konstruktorów
- LList to największy zbiór termów zbudowanych z LNil i LCons i zawierający nieskończone termy
- nie obowiązuje zasada indukcji
- kostruktory są różnowartościowe i różne (można używać taktyk injection i discriminate)

Typ leniwych list — `LList`

```
CoInductive LList (A:Set) :Set :=
   LNil : LList A
|  LCons : A -> LList A -> LList A
```

- termy zbudowane tylko z konstruktorów
- `LList` to największy zbiór termów zbudowanych z `LNil` i `LCons` i zawierający nieskończone termy
- nie obowiązuje zasada indukcji
- kostruktory są różnowartościowe i różne (można używać taktyk `injection` i `discriminate`)

Typ leniwych list — `LList`

```
CoInductive LList (A:Set) :Set :=
   LNil : LList A
|  LCons : A -> LList A -> LList A
```

- termy zbudowane tylko z konstruktorów
- `LList` to największy zbiór termów zbudowanych z `LNil` i `LCons` i zawierający nieskończone termy
- nie obowiązuje zasada indukcji
- kostruktory są różnowartościowe i różne (można używać taktyk `injection` i `discriminate`)

Typ leniwych list — `LList`

```
CoInductive LList (A:Set) :Set :=
   LNil : LList A
|  LCons : A -> LList A -> LList A
```

- termy zbudowane tylko z konstruktorów
- `LList` to największy zbiór termów zbudowanych z `LNil` i `LCons` i zawierający nieskończone termy
- nie obowiązuje zasada indukcji
- kostruktory są różnowartościowe i różne (można używać taktyk `injection` i `discriminate`)

Typ leniwych list — `LList`

```
CoInductive LList (A:Set) :Set :=
   LNil : LList A
|  LCons : A -> LList A -> LList A
```

- termy zbudowane tylko z konstruktorów
- `LList` to największy zbiór termów zbudowanych z `LNil` i `LCons` i zawierający nieskończone termy
- nie obowiązuje zasada indukcji
- kostruktory są różnowartościowe i różne (można używać taktyk `injection` i `discriminate`)

Typ leniwych drzew binarnych — LTree

```
CoInductive LTree (A:Set) :Set :=
   LLeaf : LTree A
|  LBin : A -> LTree A -> LTree A
```

- drzewa skończone i nieskończone
- tylko niektóre gałęzie mogą być nieskończone

Typ leniwych drzew binarnych — LTree

```
CoInductive LTree (A:Set) :Set :=
   LLeaf : LTree A
|  LBin : A -> LTree A -> LTree A
```

- drzewa skończone i nieskończone
- tylko niektóre gałęzie mogą być nieskończone

Typ leniwych drzew binarnych — LTree

```
CoInductive LTree (A:Set) :Set :=
   LLeaf : LTree A
|  LBin : A -> LTree A -> LTree A
```

- drzewa skończone i nieskończone
- tylko niektóre gałęzie mogą być nieskończone

Typ strumieni — Stream

```
CoInductive Stream (A:Set) :Set :=
  Cons : A -> Stream A -> Stream A
```

- nie ma skończonych strumieni
- każdy strumień jest postaci `Cons a l`

Typ strumieni — Stream

```
CoInductive Stream (A:Set) :Set :=
  Cons : A -> Stream A -> Stream A
```

- nie ma skończonych strumieni
- każdy strumień jest postaci `Cons a l`

Typ strumieni — Stream

```
CoInductive Stream (A:Set) :Set :=
  Cons : A -> Stream A -> Stream A
```

- nie ma skończonych strumieni
- każdy strumień jest postaci `Cons a l`

Pattern-matching

```
Definition isEmpty (A:Type) (l:LList A) : Prop :=
  match l with
  | LNil => True
  | LCons a l' => False
  end.
```

```
Definition LHead (A:Type) (l:LList A) : option A :=
  match l with
  | LNil => None
  | LCons a l' => Some a
  end.
```

Pattern-matching

```
Definition isEmpty (A:Type) (l:LList A) : Prop :=
  match l with
  | LNil => True
  | LCons a l' => False
  end.
```

```
Definition LHead (A:Type) (l:LList A) : option A :=
  match l with
  | LNil => None
  | LCons a l' => Some a
  end.
```

Pattern-matching

```
Fixpoint LNth (A:Type) (n:nat) (l:LList A) {struct n} :
option A :=
  match l with
  | LNil => None
  | LCons a l' => match n with
                  | O => Some a
                  | S p => LNth p l'
                  end
  end.
```

```
Eval compute in (LNth 2 (LCons 4 (LCons 3 (LCons 90 LNil)))).
  = Some 90 : option nat
```

Pattern-matching

```
Fixpoint LNth (A:Type) (n:nat) (l:LList A) {struct n} :
option A :=
  match l with
  | LNil => None
  | LCons a l' => match n with
                  | O => Some a
                  | S p => LNth p l'
                  end
  end.
```

```
Eval compute in (LNth 2 (LCons 4 (LCons 3 (LCons 90 LNil)))).
  = Some 90 : option nat
```

Budowanie obiektów nieskończonych

W sposób skończony będziemy reprezentować niektóre obiekty nieskończone... Poniżej nieudana próba:

```
Fixpoint from (n:nat) {struct n} : LList nat :=
  Lcons n (from (S n)).
```

Powód: wywołanie rekurencyjne `from` nie jest zaplikowane do termu strukturalnie mniejszego od `n`. Udana definicja to:

```
CoFixpoint from (n:nat) : LList nat := LCons n (from (S n)).
```

```
Definition Nats : LList nat := from 0.
```

Funkcje korekurencyjne — wstęp

- wszystkie obliczenia w Coqu muszą być skończone,
- stąd nie można rozważać funkcji "przechodzących" po termach typu koindukcyjnego,
- za to można rozważać funkcje budujące wynik w typie koindukcyjnym,
- wynik może być nieskończony, ale obejrzenie jego skończonej części powinno zająć tylko skończenie wiele czasu,
- funkcje rekurencyjne "konsumują" wartości typów indukcyjnych, funkcje korekurencyjne "produkują" wartości typów koindukcyjnych; oba rodzaje funkcji mają swoje "guard conditions" pilnujące wywołań rekurencyjnych.

Funkcje korekurencyjne — wstęp

- wszystkie obliczenia w Coqu muszą być skończone,
- stąd nie można rozważać funkcji "przechodzących" po termach typu koindukcyjnego,
- za to można rozważać funkcje budujące wynik w typie koindukcyjnym,
- wynik może być nieskończony, ale obejrzenie jego skończonej części powinno zająć tylko skończenie wiele czasu,
- funkcje rekurencyjne "konsumują" wartości typów indukcyjnych, funkcje korekurencyjne "produkują" wartości typów koindukcyjnych; oba rodzaje funkcji mają swoje "guard conditions" pilnujące wywołań rekurencyjnych.

Funkcje korekurencyjne — wstęp

- wszystkie obliczenia w Coqu muszą być skończone,
- stąd nie można rozważać funkcji "przechodzących" po termach typu koindukcyjnego,
- za to można rozważać funkcje budujące wynik w typie koindukcyjnym,
- wynik może być nieskończony, ale obejrzenie jego skończonej części powinno zająć tylko skończenie wiele czasu,
- funkcje rekurencyjne "konsumują" wartości typów indukcyjnych, funkcje korekurencyjne "produkują" wartości typów koindukcyjnych; oba rodzaje funkcji mają swoje "guard conditions" pilnujące wywołań rekurencyjnych.

Funkcje korekurencyjne — wstęp

- wszystkie obliczenia w Coqu muszą być skończone,
- stąd nie można rozważać funkcji "przechodzących" po termach typu koindukcyjnego,
- za to można rozważać funkcje budujące wynik w typie koindukcyjnym,
- wynik może być nieskończony, ale obejrzenie jego skończonej części powinno zająć tylko skończenie wiele czasu,
- funkcje rekurencyjne "konsumują" wartości typów indukcyjnych, funkcje korekurencyjne "produkują" wartości typów koindukcyjnych; oba rodzaje funkcji mają swoje "guard conditions" pilnujące wywołań rekurencyjnych.

Funkcje korekurencyjne — wstęp

- wszystkie obliczenia w Coqu muszą być skończone,
- stąd nie można rozważać funkcji "przechodzących" po termach typu koindukcyjnego,
- za to można rozważać funkcje budujące wynik w typie koindukcyjnym,
- wynik może być nieskończony, ale obejrzenie jego skończonej części powinno zająć tylko skończenie wiele czasu,
- funkcje rekurencyjne "konsumują" wartości typów indukcyjnych, funkcje korekurencyjne "produkują" wartości typów koindukcyjnych; oba rodzaje funkcji mają swoje "guard conditions" pilnujące wywołań rekurencyjnych.

Funkcje korekurencyjne — guard condition

Definicja przez `cofixpoint` jest poprawna jesli każde wywołanie rekurencyjne jest jednym z argumentów konstruktora typu koindukcyjnego.

- podobieństwo: w leniwych językach programowania konstruktory nie ewaluują swoich argumentów
- jeśli obiekty typów koindukcyjnych są oglądane używając pattern-matchingu to guard condition zapewnia, że każde wywołanie korekurencyjnej funkcji produkuje w skończonym czasie choć jeden konstruktor,
- funkcja rekurencyjna się rozwinie jeśli zostanie zaaplikowana do termu z konstruktorem w głowie; funkcja korekurencyjna się rozwinie tylko jeśli jest argumentem dla jakiegoś pattern-matchingu

Funkcje korekurencyjne — guard condition

Definicja przez `cofixpoint` jest poprawna jesli każde wywołanie rekurencyjne jest jednym z argumentów konstruktora typu koindukcyjnego.

- podobieństwo: w leniwych językach programowania konstruktory nie ewaluują swoich argumentów
- jeśli obiekty typów koindukcyjnych są oglądane używając pattern-matchingu to guard condition zapewnia, że każde wywołanie korekurencyjnej funkcji produkuje w skończonym czasie choć jeden konstruktor,
- funkcja rekurencyjna się rozwinie jeśli zostanie zaaplikowana do termu z konstruktorem w głowie; funkcja korekurencyjna się rozwinie tylko jeśli jest argumentem dla jakiegoś pattern-matchingu

Funkcje korekurencyjne — guard condition

Definicja przez `cofixpoint` jest poprawna jesli każde wywołanie rekurencyjne jest jednym z argumentów konstruktora typu koindukcyjnego.

- podobieństwo: w leniwych językach programowania konstruktory nie ewaluują swoich argumentów
- jeśli obiekty typów koindukcyjnych są oglądane używając pattern-matchingu to guard condition zapewnia, że każde wywołanie korekurencyjnej funkcji produkuje w skończonym czasie choć jeden konstruktor,
- funkcja rekurencyjna się rozwinie jeśli zostanie zaaplikowana do termu z konstruktorem w głowie; funkcja korekurencyjna się rozwinie tylko jeśli jest argumentem dla jakiegoś pattern-matchingu

Funkcje korekurencyjne — guard condition

Definicja przez `cofixpoint` jest poprawna jesli każde wywołanie rekurencyjne jest jednym z argumentów konstruktora typu koindukcyjnego.

- podobieństwo: w leniwych językach programowania konstruktory nie ewaluują swoich argumentów
- jeśli obiekty typów koindukcyjnych są oglądane używając pattern-matchingu to guard condition zapewnia, że każde wywołanie korekurencyjnej funkcji produkuje w skończonym czasie choć jeden konstruktor,
- funkcja rekurencyjna się rozwinie jeśli zostanie zaaplikowana do termu z konstruktorem w głowie; funkcja korekurencyjna się rozwinie tylko jeśli jest argumentem dla jakiegoś pattern-matchingu

Przykłady

```
Eval simpl in (from 3).
  = from 3 : LLIst nat

Eval simpl in (LHead (LTail (from 3))).
  = Some 4 : option nat

CoFixpoint forever (A:Type)(a:A):LList A:=LCons a (forever a).

CoFixpoint LAppend (A:Type) (u v:LList A) : LList A :=
  match u with
  | LNil => v
  | LCons a u' => LCons a (LAppend u' v)
  end.

Eval compute in (LNth 123 (LAppend (forever 33) Nats)).
  = Some 33 : option nat

Eval compute in
  (LNth 123 (LAppend (LCons 0 (LCons 1 (LCons 2 LNil))) Nats)).
  = Some 120 : option nat
```

Przykłady

```
Eval simpl in (from 3).
  = from 3 : LLIst nat

Eval simpl in (LHead (LTail (from 3))).
  = Some 4 : option nat

CoFixpoint forever (A:Type)(a:A):LList A:=LCons a (forever a).

CoFixpoint LAppend (A:Type) (u v:LList A) : LList A :=
  match u with
  | LNil => v
  | LCons a u' => LCons a (LAppend u' v)
  end.

Eval compute in (LNth 123 (LAppend (forever 33) Nats)).
  = Some 33 : option nat

Eval compute in
  (LNth 123 (LAppend (LCons 0 (LCons 1 (LCons 2 LNil))) Nats)).
  = Some 120 : option nat
```

Przykłady

```
Eval simpl in (from 3).
  = from 3 : LLIst nat

Eval simpl in (LHead (LTail (from 3))).
  = Some 4 : option nat

CoFixpoint forever (A:Type)(a:A):LList A:=LCons a (forever a).

CoFixpoint LAppend (A:Type) (u v:LList A) : LList A :=
  match u with
  | LNil => v
  | LCons a u' => LCons a (LAppend u' v)
  end.

Eval compute in (LNth 123 (LAppend (forever 33) Nats)).
  = Some 33 : option nat

Eval compute in
  (LNth 123 (LAppend (LCons 0 (LCons 1 (LCons 2 LNil))) Nats)).
  = Some 120 : option nat
```

Niepoprawne definicje przez cofixpoint

```
CoFixpoint filter (A:Set) (p: A->bool) (l:LList A) : LList A
:=
  match l with
  | LNil => LNil
  | LCons a l' => if p then LCons a (filter p l')
                       else (filter p l')   end.
```

```
LHead (filter (fun p:nat =>
                 match p with 0 => true | S n => false end)
              (from 1))
```

spowodowałoby nieskończone obliczenie

Niepoprawne definicje przez cofixpoint

```
CoFixpoint filter (A:Set) (p: A->bool) (l:LList A) : LList A
:=
  match l with
  | LNil => LNil
  | LCons a l' => if p then LCons a (filter p l')
                       else (filter p l')   end.
```

```
LHead (filter (fun p:nat =>
                  match p with 0 => true | S n => false end)
              (from 1))
```

spowodowałoby nieskończone obliczenie

Rozwijanie i lematy dekompozycji

```
Definition LList_decompose (A:Type) (l:LList A) : LList A :=
  match l with
  | LNil => LNil
  | LCons a l' => LCons a l'
  end.

Eval simpl in (LList_decompose (forever 33)).
 = LCons 33 (forever 33) : LList nat
```

```
Lemma LList_decomposition : forall (A:Type) (l:LList A), l =
LList_decompose l.
Proof.
 intros A l; case l; trivial.
Qed.
```

Rozwijanie i lematy dekompozycji

```
Definition LList_decompose (A:Type) (l:LList A) : LList A :=
  match l with
  | LNil => LNil
  | LCons a l' => LCons a l'
  end.

Eval simpl in (LList_decompose (forever 33)).
 = LCons 33 (forever 33) : LList nat


Lemma LList_decomposition : forall (A:Type) (l:LList A), l =
LList_decompose l.
Proof.
 intros A l; case l; trivial.
Qed.
```

Dowód używający dekompozycji

```
Ltac LList_unfold term := apply trans_equal with
   (1 := LList_decomposition term).

Lemma LAppend_LNil : forall (A:Type) (v:LList A),
      LAppend LNil v = v.
Proof.
 intros A v.
 LList_unfold (LAppend LNil v).
 case v; simpl in |- *; auto.
Qed.
```

Predykaty indukcyjne po koindukcyjnych typach

```
Inductive Finite (A:Type) : LList A -> Prop :=
 | Finite_LNil : Finite LNil
 | Finite_LCons : forall (a:A) (l:LList A), Finite l -> Finite
(LCons a l).

Hint Resolve Finite_LNil Finite_LCons: llists.

Remark one_two_three : Finite (LCons 1 (LCons 2 (LCons 3 LNil))).
Proof.
 auto with llists.
Qed.

Theorem Finite_of_LCons :
 forall (A:Type) (a:A) (l:LList A), Finite (LCons a l) -> Finite
l.
Proof.
 intros A a l H; inversion H; assumption.
Qed.
```

Predykaty indukcyjne po koindukcyjnych typach

```
Inductive Finite (A:Type) : LList A -> Prop :=
 | Finite_LNil : Finite LNil
 | Finite_LCons : forall (a:A) (l:LList A), Finite l -> Finite
(LCons a l).

Hint Resolve Finite_LNil Finite_LCons: llists.

Remark one_two_three : Finite (LCons 1 (LCons 2 (LCons 3 LNil))).
Proof.
 auto with llists.
Qed.

Theorem Finite_of_LCons :
 forall (A:Type) (a:A) (l:LList A), Finite (LCons a l) -> Finite
l.
Proof.
 intros A a l H; inversion H; assumption.
Qed.
```

Predykaty indukcyjne po koindukcyjnych typach

```
Inductive Finite (A:Type) : LList A -> Prop :=
 | Finite_LNil : Finite LNil
 | Finite_LCons : forall (a:A) (l:LList A), Finite l -> Finite
(LCons a l).

Hint Resolve Finite_LNil Finite_LCons: llists.

Remark one_two_three : Finite (LCons 1 (LCons 2 (LCons 3 LNil)))
Proof.
 auto with llists.
Qed.

Theorem Finite_of_LCons :
 forall (A:Type) (a:A) (l:LList A), Finite (LCons a l) -> Finite
l.
Proof.
 intros A a l H; inversion H; assumption.
Qed.
```

Predykaty koindukcyjne

```
CoInductive Infinite (A:Type) : LList A -> Prop :=
 Infinite_LCons :
   forall (a:A) (l:LList A), Infinite l -> Infinite (LCons a
l).

Hint Resolve Infinite_LCons: llists.
```

Chcemy dowieść, że `forall n:nat, Infinite (from n)`. Pomocniczo potrzebujemy dekompozycji dla `from`:

```
Lemma from_unfold : forall n:nat, from n = LCons n (from (S
n)).
Proof.
 intro n.
 LList_unfold (from n).
 simpl in |- *; trivial.
Qed.
```

Predykaty koindukcyjne

```
CoInductive Infinite (A:Type) : LList A -> Prop :=
 Infinite_LCons :
   forall (a:A) (l:LList A), Infinite l -> Infinite (LCons a
l).

Hint Resolve Infinite_LCons: llists.
```

Chcemy dowieść, że `forall n:nat, Infinite (from n)`. Pomocniczo potrzebujemy dekompozycji dla `from`:

```
Lemma from_unfold : forall n:nat, from n = LCons n (from (S
n)).
Proof.
 intro n.
 LList_unfold (from n).
 simpl in |- *; trivial.
Qed.
```

Dowód, że forall n:nat, Infinite (from n)

Dowodem będzie korekurencyjna funkcja — punkt stały F_from:

```
Definition F_from :
 (forall n:nat, Infinite (from n)) -> forall n:nat, Infinite
(from n).

 intros H n; rewrite (from_unfold n).
 constructor; auto.
Defined.
```

```
Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof (cofix H : forall n:nat, Infinite (from n) := F_from H).
```

```
Lemma from_Infinite : forall n:nat, Infinite (from n).
Proof.
 cofix H.
 intro n; rewrite (from_unfold n).
 constructor; auto.
Qed.
```

Dowód, że forall n:nat, Infinite (from n)

Dowodem będzie korekurencyjna funkcja — punkt stały F_from:

```
Definition F_from :
 (forall n:nat, Infinite (from n)) -> forall n:nat, Infinite
(from n).

 intros H n; rewrite (from_unfold n).
 constructor; auto.
Defined.


Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof (cofix H : forall n:nat, Infinite (from n) := F_from H).

Lemma from_Infinite : forall n:nat, Infinite (from n).
Proof.
 cofix H.
 intro n; rewrite (from_unfold n).
 constructor; auto.
Qed.
```

Dowód, że forall n:nat, Infinite (from n)

Dowodem będzie korekurencyjna funkcja — punkt stały F_from:

```
Definition F_from :
 (forall n:nat, Infinite (from n)) -> forall n:nat, Infinite
(from n).

 intros H n; rewrite (from_unfold n).
 constructor; auto.
Defined.


Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof (cofix H : forall n:nat, Infinite (from n) := F_from H).

Lemma from_Infinite : forall n:nat, Infinite (from n).
Proof.
 cofix H.
 intro n; rewrite (from_unfold n).
 constructor; auto.
Qed.
```

Zły dowód, że `forall n:nat, Infinite (from n)`

```
Lemma from_Infinite_buggy : forall n:nat, Infinite (from n).
Proof.
 cofix H.
 auto with llists.
 Qed.
```

Error: Recursive definition of "H" is ill-formed.
In environment
H: ∀ n:nat, Infinite (from n)
ungarded recursive call in H

Ratunek: użycie komendy `Guarded` podczas dowodzenia, powoduje sprawdzenie czy na obecnym etapie dowodu "guard condition" jest spełniony

Zły dowód, że `forall n:nat, Infinite (from n)`

```
Lemma from_Infinite_buggy : forall n:nat, Infinite (from n).
Proof.
 cofix H.
 auto with llists.
 Qed.
```

Error: Recursive definition of "H" is ill-formed.
In environment
H: ∀ n:nat, Infinite (from n)
ungarded recursive call in H

Ratunek: użycie komendy `Guarded` podczas dowodzenia, powoduje sprawdzenie czy na obecnym etapie dowodu "guard condition" jest spełniony

Eliminacja założeń koindukcyjnych

Taktyki `case` i `inversion`:

```
Lemma LNil_not_Infinite : forall A:Type, ~ Infinite (@LNil A).
Proof.
 intros A H; inversion H.
Qed.
```

Równość obiektów koindukcyjnych

Równość eq jest dobra jeśli skończenie wiele rozwinięć pozwala dojść do konwertowalnych termów. Są przykłady gdy tak nie jest:

```
Lemma Lappend_of_Infinite_0 :
 forall (A:Type) (u:LList A), Infinite u -> forall v:LList A,
u = LAppend u v.
```

Równość eq jest za mocna, trzeba wprowadzić słabszy predykat

Równość obiektów koindukcyjnych

Równość eq jest dobra jeśli skończenie wiele rozwinięć pozwala dojść do konwertowalnych termów. Są przykłady gdy tak nie jest:

```
Lemma Lappend_of_Infinite_0 :
 forall (A:Type) (u:LList A), Infinite u -> forall v:LList A,
u = LAppend u v.
```

Równość eq jest za mocna, trzeba wprowadzić słabszy predykat

Bisymularność

```
CoInductive bisimilar (A:Type) : LList A -> LList A -> Prop
:=
 | bisim0 : bisimilar LNil LNil
 | bisim1 :
      forall (a:A) (l l':LList A),
        bisimilar l l' -> bisimilar (LCons a l) (LCons a l').

Hint Resolve bisim0 bisim1: llists.
```

Bisymulacja

```
Definition bisimulation (A:Type) (R:LList A -> LList A -> Prop)
:=
 forall l1 l2:LList A,
  R l1 l2 ->
  match l1 with
  | LNil => l2 = LNil
  | LCons a l'1 =>
      match l2 with
      | LNil => False
      | LCons b l'2 => a = b ∧ R l'1 l'2
      end
end.
```

Park principle

Bisymularność to największa relacja zachodząca na `Lnil` i zamknięta na `LCons`. *Bisymulacja* to relacja o takich własnościach ale nie koniecznie największa. Stąd wynika poprawność "Park principle"

```
Theorem park_principle :
 forall (A:Type) (R:LList A -> LList A -> Prop),
  bisimulation R -> forall l1 l2:LList A, R l1 l2 ->
    bisimilar l1 l2.
```

Koindukcyjna semantyka operacyjna dla języka while-programów

Zamodelujemy możliwość zapętlenia używając typów koindukcyjnych; wykonanie programu będzie skończone albo nieskończone.

```
Definition var := nat.

Definition vars := var → nat.
Definition set (vs : vars) (v : var) (n : nat) : vars :=
  fun v' ⇒ if beq_nat v v' then n else vs v'.
```

Wyrażenia

```
Inductive exp : Set :=
| Const : nat → exp
| Var : var → exp
| Plus : exp → exp → exp.
```

```
Fixpoint evalExp (vs : vars) (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Var v ⇒ vs v
    | Plus e1 e2 ⇒ evalExp vs e1 + evalExp vs e2
  end.
```

Wyrażenia

```
Inductive exp : Set :=
| Const : nat → exp
| Var : var → exp
| Plus : exp → exp → exp.
```

```
Fixpoint evalExp (vs : vars) (e : exp) : nat :=
  match e with
    | Const n ⇒ n
    | Var v ⇒ vs v
    | Plus e1 e2 ⇒ evalExp vs e1 + evalExp vs e2
  end.
```

Instrukcje

```
Inductive cmd : Set :=
| Assign : var → exp → cmd
| Seq : cmd → cmd → cmd
| While : exp → cmd → cmd.
```

Semantyka wykonania

Program nieterminujący w danym stanie początkowym jest związany z każdym stanem końcowym.

```
CoInductive evalCmd : vars → cmd → vars → Prop :=
| EvalAssign : ∀ vs v e, evalCmd vs (Assign v e) (set vs v (evalExp vs e))
| EvalSeq : ∀ vs1 vs2 vs3 c1 c2, evalCmd vs1 c1 vs2
  → evalCmd vs2 c2 vs3
  → evalCmd vs1 (Seq c1 c2) vs3
| EvalWhileFalse : ∀ vs e c, evalExp vs e = 0
  → evalCmd vs (While e c) vs
| EvalWhileTrue : ∀ vs1 vs2 vs3 e c, evalExp vs1 e ≠ 0
  → evalCmd vs1 c vs2
  → evalCmd vs2 (While e c) vs3
  → evalCmd vs1 (While e c) vs3.
```

Bisymulacja dla evalCmd

```
Section evalCmd_coind.
  Variable R : vars → cmd → vars → Prop.

  Hypothesis AssignCase : ∀ vs1 vs2 v e, R vs1 (Assign v e) vs2
    → vs2 = set vs1 v (evalExp vs1 e).

  Hypothesis SeqCase : ∀ vs1 vs3 c1 c2, R vs1 (Seq c1 c2) vs3
    → ∃ vs2, R vs1 c1 vs2 ∧ R vs2 c2 vs3.

  Hypothesis WhileCase : ∀ vs1 vs3 e c, R vs1 (While e c) vs3
    → (evalExp vs1 e = 0 ∧ vs3 = vs1)
    ∨ ∃ vs2, evalExp vs1 e ≠ 0 ∧ R vs1 c vs2 ∧ R vs2 (While e c)
vs3.
```

Bisymulacja dla evalCmd cd.

```
Theorem evalCmd_coind : ∀ vs1 c vs2, R vs1 c vs2 → evalCmd vs1
c vs2.
  cofix; intros; destruct c.
  rewrite (AssignCase H); constructor.
  destruct (SeqCase H) as [? [? ?]]; econstructor; eauto.
  destruct (WhileCase H) as [[? ?] | [? [? [? ?]]]]; subst;
econstructor; eauto.
Qed.
End evalCmd_coind.
```

Optymalizacja

```
Fixpoint optExp (e : exp) : exp :=
  match e with
    | Plus (Const 0) e ⇒ optExp e
    | Plus e1 e2 ⇒ Plus (optExp e1) (optExp e2)
    | _ ⇒ e
  end.
```

```
Fixpoint optCmd (c : cmd) : cmd :=
  match c with
    | Assign v e ⇒ Assign v (optExp e)
    | Seq c1 c2 ⇒ Seq (optCmd c1) (optCmd c2)
    | While e c ⇒ While (optExp e) (optCmd c)
  end.
```

Optymalizacja

```
Fixpoint optExp (e : exp) : exp :=
  match e with
    | Plus (Const 0) e ⇒ optExp e
    | Plus e1 e2 ⇒ Plus (optExp e1) (optExp e2)
    | _ ⇒ e
  end.
```

```
Fixpoint optCmd (c : cmd) : cmd :=
  match c with
    | Assign v e ⇒ Assign v (optExp e)
    | Seq c1 c2 ⇒ Seq (optCmd c1) (optCmd c2)
    | While e c ⇒ While (optExp e) (optCmd c)
  end.
```

Poprawnośc optymalizacji dla wyrażeń

```
Lemma optExp_correct : ∀ vs e, evalExp vs (optExp e) = evalExp vs e.
  induction e; crush;
    repeat (match goal with
                | [ ⊢ context[match ?E with Const _ ⇒ _ | _ ⇒ _
end] ] ⇒ destruct E
                | [ ⊢ context[match ?E with O ⇒ _ | S _ ⇒ _ end] ]
⇒ destruct E
            end; crush).
Qed.

Hint Rewrite optExp_correct.
```

Poprawność optymalizacji dla instrukcji

```
Ltac finisher := match goal with
                   | [ H : evalCmd _ _ _ ⊢ _ ] ⇒ ((inversion H; [])
                     || (inversion H; [||])); subst
                 end; crush; eauto 10.

Lemma optCmd_correct1 : ∀ vs1 c vs2, evalCmd vs1 c vs2
  → evalCmd vs1 (optCmd c) vs2.
  intros; apply (evalCmd_coind (fun vs1 c' vs2 ⇒ ∃ c, evalCmd
vs1 c vs2
    ∧ c' = optCmd c)); eauto; crush;
    match goal with
      | [ H : _ = optCmd ?E ⊢ _ ] ⇒ destruct E; simpl in *;
discriminate
        || injection H; intros; subst
    end; finisher.
Qed.
```

Poprawność optymalizacji dla instrukcji

```
Ltac finisher := match goal with
                   | [ H : evalCmd _ _ _ ⊢ _ ] ⇒ ((inversion H; [])
                     || (inversion H; [||])); subst
                 end; crush; eauto 10.

Lemma optCmd_correct1 : ∀ vs1 c vs2, evalCmd vs1 c vs2
  → evalCmd vs1 (optCmd c) vs2.
  intros; apply (evalCmd_coind (fun vs1 c' vs2 ⇒ ∃ c, evalCmd
vs1 c vs2
    ∧ c' = optCmd c)); eauto; crush;
    match goal with
      | [ H : _ = optCmd ?E ⊢ _ ] ⇒ destruct E; simpl in *;
discriminate
        || injection H; intros; subst
    end; finisher.
Qed.
```

Poprawność optymalizacji dla instrukcji cd.

```
Lemma optCmd_correct2 : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  → evalCmd vs1 c vs2.
  intros; apply (evalCmd_coind (fun vs1 c vs2 ⇒ evalCmd vs1
(optCmd c) vs2));
    crush; finisher.
Qed.
```

```
Theorem optCmd_correct : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  ↔ evalCmd vs1 c vs2.
  intuition; apply optCmd_correct1 || apply optCmd_correct2;
assumption.
Qed.
```

Poprawność optymalizacji dla instrukcji cd.

```
Lemma optCmd_correct2 : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  → evalCmd vs1 c vs2.
  intros; apply (evalCmd_coind (fun vs1 c vs2 ⇒ evalCmd vs1
(optCmd c) vs2));
    crush; finisher.
Qed.
```

```
Theorem optCmd_correct : ∀ vs1 c vs2, evalCmd vs1 (optCmd c) vs2
  ↔ evalCmd vs1 c vs2.
  intuition; apply optCmd_correct1 || apply optCmd_correct2;
assumption.
Qed.
```