Coinductive types in Coq

Daria Walukiewicz-Chrząszcz

30 april 2019

In Coq nontermination has to be avoided

General recursion will make Coq inconsistent

```
Fixpoint bad (u : unit) : P := bad u.
```

Fixpoint definition has its "guard condition" (recursive calls has to be done on structurally smaller terms) and it reduces only when `aki` (the argument one does recursion on) starts with a constructor:

```
(Fix f a1 ... aki) -> ti a1 ... aki
```

`Fix` and `Case` reductions are called together iota reduction

In Coq nontermination has to be avoided

General recursion will make Coq inconsistent

```
Fixpoint bad (u : unit) : P := bad u.
```

Fixpoint definition has its "guard condition" (recursive calls has to be done on structurally smaller terms) and it reduces only when `aki` (the argument one does recursion on) starts with a constructor:

```
(Fix f a1 ... aki) -> ti a1 ... aki
```

`Fix` and `Case` reductions are called together iota reduction

In Coq nontermination has to be avoided

General recursion will make Coq inconsistent

```
Fixpoint bad (u : unit) : P := bad u.
```

Fixpoint definition has its “guard condition” (recursive calls has to be done on structurally smaller terms) and it reduces only when `aki` (the argument one does recursion on) starts with a constructor:

```
(Fix f a1 ... aki) -> ti a1 ... aki
```

`Fix` and `Case` reductions are called together iota reduction

In Coq nontermination has to be avoided

General recursion will make Coq inconsistent

```
Fixpoint bad (u : unit) : P := bad u.
```

Fixpoint definition has its "guard condition" (recursive calls has to be done on structurally smaller terms) and it reduces only when `aki` (the argument one does recursion on) starts with a constructor:

```
(Fix f a1 ... aki) -> ti a1 ... aki
```

`Fix` and `Case` reductions are called together iota reduction

Lazy lists — LList

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

- terms built from constructors
- LList is the greatest set of terms built from LNil i LCons containing infinite and finite terms
- induction principle does not hold
- constructors are injective and distinct (one may use tactics injection and discriminate)

Lazy lists — LList

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

- terms built from constructors
- LList is the greatest set of terms built from LNil i LCons containing infinite and finite terms
- induction principle does not hold
- constructors are injective and distinct (one may use tactics injection and discriminate)

Lazy lists — LList

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

- terms built from constructors
- LList is the greatest set of terms built from LNil i LCons containing infinite and finite terms
- induction principle does not hold
- constructors are injective and distinct (one may use tactics injection and discriminate)

Lazy lists — LList

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

- terms built from constructors
- `LList` is the greatest set of terms built from `LNil` i `LCons` containing infinite and finite terms
- induction principle does not hold
- constructors are injective and distinct (one may use tactics `injection` and `discriminate`)

Lazy lists — LList

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

- terms built from constructors
- `LList` is the greatest set of terms built from `LNil` i `LCons` containing infinite and finite terms
- induction principle does not hold
- constructors are injective and distinct (one may use tactics `injection` and `discriminate`)

Lazy trees — LTree

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

- finite and infinite trees
- some branches can be infinite

Lazy trees — LTree

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

- finite and infinite trees
- some branches can be infinite

Lazy trees — LTree

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

- finite and infinite trees
- some branches can be infinite

Streams — Stream

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

- there are no finite streams
- every stream is of the form `Cons a l`

Streams — Stream

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

- there are no finite streams
- every stream is of the form `Cons a l`

Streams — Stream

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

- there are no finite streams
- every stream is of the form `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
```

Building infinite objects

Goal: to represent infinite objects in a finite way. Failed attempt:

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

Reason: recursive call `from` is not applied to structurally smaller argument. Successful attempt:

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

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

Building infinite objects

Goal: to represent infinite objects in a finite way. Failed attempt:

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

Reason: recursive call from is not applied to structurally smaller argument. Successful attempt:

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

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

Building infinite objects

Goal: to represent infinite objects in a finite way. Failed attempt:

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

Reason: recursive call `from` is not applied to structurally smaller argument. Successful attempt:

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

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

Building infinite objects

Goal: to represent infinite objects in a finite way. Failed attempt:

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

Reason: recursive call `from` is not applied to structurally smaller argument. Successful attempt:

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

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

Corecursive functions — introduction

- all computations in Coq are finite,
- recursive function consumes values of an inductive type,
- corecursive function produces values in a coinductive type,
- result may be infinite, but its every finite aproximation should be computable in finite time,
- corecursive functions have its "guard conditions".

Corecursive functions — introduction

- all computations in Coq are finite,
- recursive function consumes values of an inductive type,
- corecursive function produces values in a coinductive type,
- result may be infinite, but its every finite aproximation should be computable in finite time,
- corecursive functions have its "guard conditions".

Corecursive functions — introduction

- all computations in Coq are finite,
- recursive function consumes values of an inductive type,
- corecursive function produces values in a coinductive type,
- result may be infinite, but its every finite aproximation should be computable in finite time,
- corecursive functions have its "guard conditions".

Corecursive functions — introduction

- all computations in Coq are finite,
- recursive function consumes values of an inductive type,
- corecursive function produces values in a coinductive type,
- result may be infinite, but its every finite aproximation should be computable in finite time,
- corecursive functions have its "guard conditions".

Corecursive functions — introduction

- all computations in Coq are finite,
- recursive function consumes values of an inductive type,
- corecursive function produces values in a coinductive type,
- result may be infinite, but its every finite aproximation should be computable in finite time,
- corecursive functions have its "guard conditions".

Corecursive functions — guard condition

Definition by `cofixpoint` is correct if every (co)recursive call is one of the arguments of some constructor of a coniductive type.

- similarity: in lazy programming languages constructors do not evaluate its arguments
- if coinductive values are matched against patterns, then guard condition ensures that every recursive call of a corecursive function produces in a finite time its head-constructor
- recursive function reduces when it is applied to a value with constructor in head position; corecursive function reduces when it is an argument to pattern-matching

Corecursive functions — guard condition

Definition by `cofixpoint` is correct if every (co)recursive call is one of the arguments of some constructor of a coniductive type.

- similarity: in lazy programming languages constructors do not evaluate its arguments
- if coinductive values are matched against patterns, then guard condition ensures that every recursive call of a corecursive function produces in a finite time its head-constructor
- recursive function reduces when it is applied to a value with constructor in head position; corecursive function reduces when it is an argument to pattern-matching

Corecursive functions — guard condition

Definition by `cofixpoint` is correct if every (co)recursive call is one of the arguments of some constructor of a coniductive type.

- similarity: in lazy programming languages constructors do not evaluate its arguments
- if coinductive values are matched against patterns, then guard condition ensures that every recursive call of a corecursive function produces in a finite time its head-constructor
- recursive function reduces when it is applied to a value with constructor in head position; corecursive function reduces when it is an argument to pattern-matching

Corecursive functions — guard condition

Definition by `cofixpoint` is correct if every (co)recursive call is one of the arguments of some constructor of a coniductive type.

- similarity: in lazy programming languages constructors do not evaluate its arguments
- if coinductive values are matched against patterns, then guard condition ensures that every recursive call of a corecursive function produces in a finite time its head-constructor
- recursive function reduces when it is applied to a value with constructor in head position; corecursive function reduces when it is an argument to pattern-matching

Examples

```
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
```

Examples

```
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
```

Examples

```
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
```

Incorrect definitions by cofixpoint

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

would cause an infinite computation

Incorrect definitions by cofixpoint

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

would cause an infinite computation

Decomposition lemmas

```
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.
```

Decomposition lemmas

```
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.
```

Proofs using decomposition

```
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 |- *; reflexivity.
Qed.
```

Inductive predicates on coinductive types

```
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).
```

```
Remark one_two_three : Finite (LCons 1 (LCons 2 (LCons 3 LNil))).
Proof.
 repeat constructor.
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.
```

Inductive predicates on coinductive types

```
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).

Remark one_two_three : Finite (LCons 1 (LCons 2 (LCons 3 LNil)))
Proof.
 repeat constructor.
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.
```

Inductive predicates on coinductive types

```
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).


Remark one_two_three : Finite (LCons 1 (LCons 2 (LCons 3 LNil)))
Proof.
 repeat constructor.
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.
```

Coinductive predicates

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

We want to prove that `forall n:nat, Infinite (from n)`.
We need an auxiliary decomposition lemma for `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.
```

Coinductive predicates

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

We want to prove that `forall n:nat, Infinite (from n)`.
We need an auxiliary decomposition lemma for `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.
```

Proof of forall n:nat, Infinite (from n)

The proof will be a corecursive function — the greatest fixpoint of 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; apply H.
Qed.
```

Proof of forall n:nat, Infinite (from n)

The proof will be a corecursive function — the greatest fixpoint of 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; apply H.
Qed.
```

Proof of forall n:nat, Infinite (from n)

The proof will be a corecursive function — the greatest fixpoint of 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; apply H.
Qed.
```

Wrong proof of `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

Note: you may use command `Guarded`, to check that "guard condition" is still satisfied

Wrong proof of `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

Note: you may use command `Guarded`, to check that "guard condition" is still satisfied

Elimination of coinductive assumptions

Tactics `case` and `inversion` work for coinductive types:

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

Equality of coinductive objects

Equality eq is adequate if finite number of simplification results in identical terms. There are examples when it does not hold:

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

Equality eq is too strong, one needs a weaker predicate.

Equality of coinductive objects

Equality eq is adequate if finite number of simplification results in identical terms. There are examples when it does not hold:

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

Equality eq is too strong, one needs a weaker predicate.

Bisimilarity

```
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').
```

Bisimulation

```
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

Bisimilarity is the greatest relation containing the pair `Lnil`, `LNil` and closed under application of `LCons`.

Bisimulation is any relation satisfying these closure properties. Hence:

```
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.
```

Park principle

Bisimilarity is the greatest relation containing the pair `Lnil`, `LNil` and closed under application of `LCons`.

Bisimulation is any relation satisfying these closure properties. Hence:

```
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.
```

Coinductive operational semantics for while-programs (example from CPDT)

Nonterminating (and terminating) programs will be modeled using coinductive types.

```
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'.
```

Expressions

```
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.
```

Expressions

```
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.
```

Instructions

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

Operational semantics

A program that does not terminate in a particular initial state is related to any final state.

```
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.
```

Operational semantics

A program that does not terminate in a particular initial state is related to any final state.

`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*.

Bisimulation for evalCmd

Section evalCmd_coind.

Variable R : vars $\rightarrow$ **cmd** $\rightarrow$ vars $\rightarrow$ Prop.

Hypothesis $AssignCase$: $\forall$ $vs1$ $vs2$ v e, R $vs1$ (Assign v e) $vs2$
$\rightarrow$ $vs2$ = set $vs1$ v (evalExp $vs1$ e).

Hypothesis $SeqCase$: $\forall$ $vs1$ $vs3$ $c1$ $c2$, R $vs1$ (Seq $c1$ $c2$) $vs3$
$\rightarrow$ $\exists$ $vs2$, R $vs1$ $c1$ $vs2$ $\wedge$ R $vs2$ $c2$ $vs3$.

Hypothesis $WhileCase$: $\forall$ $vs1$ $vs3$ e c, R $vs1$ (While e c) $vs3$
$\rightarrow$ (evalExp $vs1$ e = 0 $\wedge$ $vs3$ = $vs1$)
$\vee$ $\exists$ $vs2$, evalExp $vs1$ e $\neq$ 0 $\wedge$ R $vs1$ c $vs2$ $\wedge$ R $vs2$ (While e c)
$vs3$.

Bisimulation for evalCmd cont.

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.
```

Optimization

```
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.
```

Optimization

```
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.
```

Optimization correctness for expressions

`Lemma` optExp_correct : $\forall$ *vs* *e*, evalExp *vs* (optExp *e*) = evalExp *vs* *e*.

Optimization correctness for instructions

Lemma optCmd_correct1 : ∀ *vs1 c vs2*, **evalCmd** *vs1 c vs2*
→ **evalCmd** *vs1* (optCmd *c*) *vs2*.

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.

Optimization correctness for instructions

Lemma optCmd_correct1 : ∀ *vs1 c vs2*, **evalCmd** *vs1 c vs2*
→ **evalCmd** *vs1* (optCmd *c*) *vs2*.

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.

Optimization correctness for instructions, cont.

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