Coinductive Types in Coq

Jan Heemstra and Erik Voogd

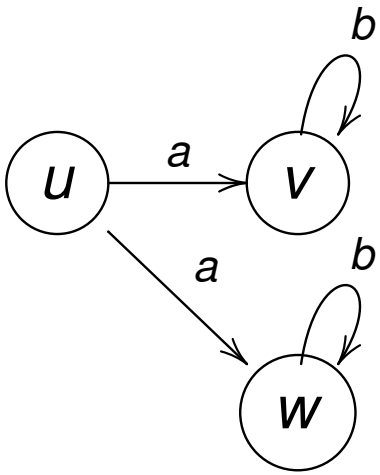
Type theory seminar

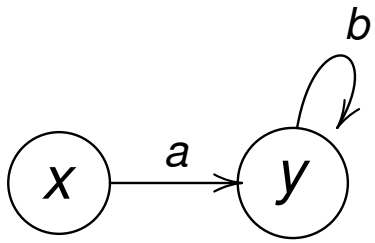
Radboud University Nijmegen

April 23, 2019

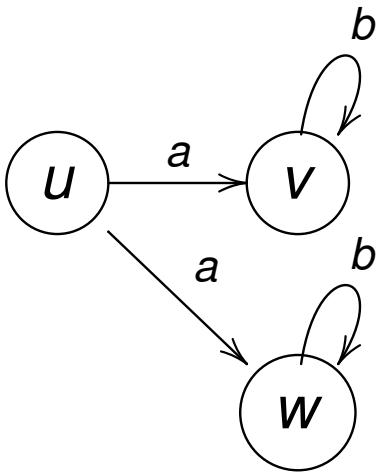
Motivation

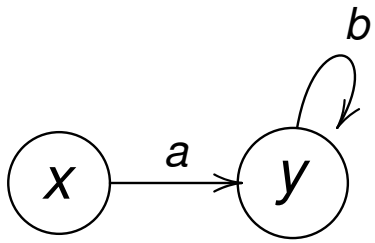
Motivation





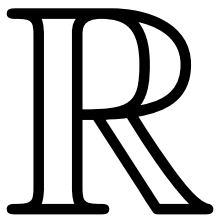
Motivation





$$\mathcal{L} = ab^{\omega}$$

Motivation



Motivation

$$\sqrt{2} = 1.01101010000010011110\ldots$$

Coinduction

Outline

- Induction and recursive functions
- Coinductive types in Coq: `CoInductive`
- Corecursive functions in Coq: `CoFixpoint`
- Unfolding techniques
- Proof demonstrations

Induction

- Inductive types such as natural numbers:

```
(* Natural numbers *)
Inductive Nat :=
| Zero : Nat
| Suc : Nat -> Nat.
```

Induction

- Inductive types such as lists:

```
Set Implicit Arguments.
Inductive List (A:Set) :=
| Nil : List A
| Cons : A -> List A -> List A.
Arguments Nil [A].
Eval compute in Cons 0 (Cons 1 (Cons 2 Nil)).
```

Induction

- Inductive types such as lists:

```
Set Implicit Arguments.
Inductive List (A:Set) :=
| Nil : List A
| Cons : A -> List A -> List A.
Arguments Nil [A].
Eval compute in Cons 0 (Cons 1 (Cons 2 Nil)).
```

- Instances of the type are finitely constructed

Induction

- Inductive types such as natural numbers:

```
(* Natural numbers *)
Inductive Nat :=
| Zero : Nat
| Suc : Nat -> Nat.
```

- Instances of the type are finitely constructed
- Recursive *function* definitions such as addition:

```
Fixpoint plus (n m:Nat) : Nat :=
 match m with
  | Zero => n
  | Suc p => Suc (plus n p)
end.
```

Induction

- Inductive types such as natural numbers:

```
(* Natural numbers *)
Inductive Nat :=
| Zero : Nat
| Suc : Nat -> Nat.
```

- Instances of the type are finitely constructed
- Recursive *function* definitions such as addition:

```
Fixpoint plus (n m:Nat) : Nat :=
 match m with
  | Zero => n
  | Suc p => Suc (plus n p)
end.
```

- Termination property: only reduced terms in recursive calls

Induction

- Inductive types such as lists:

```
Set Implicit Arguments.
Inductive List (A:Set) :=
| Nil : List A
| Cons : A -> List A -> List A.
Arguments Nil [A].
Eval compute in Cons 0 (Cons 1 (Cons 2 Nil)).
```

- Instances of the type are finitely constructed
- Recursive *function* definitions such as `map`:

```
Fixpoint map (A B:Set) (f:A->B) (s:List A) : List B :=
 match s with
| Nil => Nil
| Cons a t => Cons (f a) (map f t)
 end.
Eval compute in map plus1 (Cons 0 (Cons 41 Nil)).
```

- Termination property: only reduced terms in recursive calls
- Infinite stream: `Cons 1 (Cons 0 (Cons 1 (Cons 1 ...)))` ?

Coinductive Types (1/3)

- Example construction of a stream: $2^\omega = (2, 2, 2, \ldots)$

Coinductive Types (1/3)

- Example construction of a stream: $2^\omega = (2, 2, 2, \ldots)$
- Attempt to define streams:

```
Inductive Stream (A:Set) :=
| SCons : A -> Stream A -> Stream A.
Arguments SCons [A].
```

Coinductive Types (1/3)

- Example construction of a stream: $2^\omega = (2, 2, 2, \ldots)$
- Attempt to define streams:

```
Inductive Stream (A:Set) :=
| SCons : A -> Stream A -> Stream A.
Arguments SCons [A].
```

- Coq accepts this!
- Function for creating an infinite repeating stream:

```
Fixpoint repeat (n:nat) : Stream nat :=
SCons n (repeat n).
```

Coinductive Types (1/3)

- Example construction of a stream: $2^\omega = (2, 2, 2, \ldots)$
- Attempt to define streams:

```
Inductive Stream (A:Set) :=
| SCons : A -> Stream A -> Stream A.
Arguments SCons [A].
```

- Coq accepts this!
- Function for creating an infinite repeating stream:

```
Fixpoint repeat (n:nat) : Stream nat :=
SCons n (repeat n).
```

- Now Coq will panic:

```
Recursive definition of repeat is ill-formed.
...
Recursive call to repeat has principal argument equal to "n"
instead of a subterm of "n".
...
```

Coinductive Types (2/3)

Coinduction

to the

RESCUE

Coinductive Types (3/3)

X

```
Inductive Stream (A:Set) :=
| SCons : A -> Stream A -> Stream A.
Arguments SCons [A].

Fixpoint repeat (n:nat) : Stream nat :=
 SCons n (repeat n).
```

Coinductive Types (3/3)

```
CoInductive Stream (A:Set) :=
| SCons : A -> Stream A -> Stream A.
Arguments SCons [A].

CoFixpoint repeat (n:nat) : Stream nat :=
 SCons n (repeat n).
```

Coinductive Types (3/3)

```
CoInductive Stream (A:Set) :=
| SCons : A -> Stream A -> Stream A.
Arguments SCons [A].

CoFixpoint repeat (n:nat) : Stream nat :=
 SCons n (repeat n).
```

Both use *pattern matching* to deconstruct the type:

```
Definition head_S (A:Set) (s:Stream A) : A :=
 match s with
 | SCons a t => a
 end.
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

```
Inductive List (A:Set) :=
| Nil : List A
| Cons : A -> List A -> List A.
Arguments Nil [A].

CoFixpoint repeat (n:nat) : List nat :=
 Cons n (repeat n).
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

```
Inductive List (A:Set) :=
| Nil : List A
| Cons : A -> List A -> List A.
Arguments Nil [A].

CoFixpoint repeat (n:nat) : List nat :=
 Cons n (repeat n).
```

```
Recursive definition of repeat is ill-formed.
...
The codomain is "List nat" which should be a coinductive type.
...
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

```
CoInductive LList (A:Set) :=
| LNil : LList A
| LCons : A -> LList A -> LList A.
Arguments LNil [A].

CoFixpoint repeat (n:nat) : LList nat :=
 LCons n (repeat n).
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

(2) The **guard** condition

* *Recursive calls happen only under constructor arguments*

** *Recursive calls in constructor arguments do not appear as an argument of any function*

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

(2) The **guard** condition

* *Recursive calls happen only under constructor arguments*

** *Recursive calls in constructor arguments do not appear as an argument of any function*

```
CoFixpoint from (n:nat) : Stream nat :=
 SCons n (from (S n)).
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

(2) The **guard** condition

* *Recursive calls happen only under constructor arguments*

** *Recursive calls in constructor arguments do not appear as an argument of any function*

```
CoFixpoint from (n:nat) : Stream nat :=
 SCons n (from (S n)).
```

(Compare for example `from 0` in Coq with `[0..]` in Haskell)

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

(2) The **guard** condition

* *Recursive calls happen only under constructor arguments*

** *Recursive calls in constructor arguments do not appear as an argument of any function*

```
CoFixpoint filter (A:Set) (p:A->bool) (l:Stream A) : Stream A :=
 match l with
  SCons a m => if p a then SCons a (filter p m)
                      else (filter p m)
               end.
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

(2) The **guard** condition

* *Recursive calls happen only under constructor arguments*

** *Recursive calls in constructor arguments do not appear as an argument of any function*

```
CoFixpoint filter (A:Set) (p:A->bool) (l:Stream A) : Stream A :=
 match l with
  SCons a m => if p a then SCons a (filter p m)
                      else (filter p m)
               end.
```

```
Recursive definition of repeat is ill-formed.
...
Unguarded recursive call in "filter A p m".
...
```

Corecursive Functions

Two important conditions on CoFixpoint command:

(1) The codomain must be a coinductive type

(2) The **guard** condition

* *Recursive calls happen only under constructor arguments*

** *Recursive calls in constructor arguments do not appear as an argument of any function*

X

```
nats = SCons 0 (map (+1) nats)
```

Overview

Induction

- Type: `Inductive`
- Recursion: `Fixpoint`
- Termination property
- Structural recursion
- Initial algebra

Coinduction

- Type: `CoInductive`
- Corecursion: `CoFixpoint`
- Productivity
- Guarded *co*recursion
- Final *co*algebra

Overview

Induction

- Type: `Inductive`
- Recursion: `Fixpoint`
- Termination property
- Structural recursion
- Initial algebra

Coinduction

- Type: `CoInductive`
- Corecursion: `CoFixpoint`
- Productivity
- Guarded *co*recursion
- Final *co*algebra

The domain of a recursive function is an inductive type

$$\rightleftharpoons$$

The codomain of a corecursive function is a coinductive type

Unfolding Techniques (1/3)

- We consider the following function:

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

Unfolding Techniques (1/3)

- We consider the following function:

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

- Why would this ever be useful?

Unfolding Techniques (1/3)

- We consider the following function:

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

- Why would this ever be useful?

```
Eval simpl in LList_decompose (repeat 42).
```

Unfolding Techniques (1/3)

- We consider the following function:

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

- Why would this ever be useful?

```
Eval simpl in LList_decompose (repeat 42).
```

```
= LCons 42 (repeat 42)
:  LList nat
```

Unfolding Techniques (2/3)

- We need to prove equality:

```
Lemma LList_decomposition_lemma :
forall (A : Set) (l : LList A), (l = LList_decompose l).
```

- Which we can use in proofs:

```
Lemma repeat_unfold: forall (A : Set) (a : A),
(repeat a) = LCons a (repeat a).
Proof.
intros A l. apply LList_decomposition_lemma.
Qed.
```

Unfolding Techniques (2/3)

- We need to prove equality:

```
Lemma LList_decomposition_lemma :
forall (A : Set) (l : LList A), (l = LList_decompose l).
```

- Which we can use in proofs:

```
Lemma repeat_unfold: forall (A : Set) (a : A),
(repeat a) = LCons a (repeat a).
Proof.
intros A l. apply LList_decomposition_lemma.
Qed.
```

- Having to reapply decomposition can be time consuming

Unfolding Techniques (3/3)

- We want a function that decomposes n times.

```
Fixpoint LList_decomp_n (A : Set) (n : nat) (l : LList A) :
LList A :=
match n, l with
| O, l => l
| S m, LNil => LNil
| S m, LCons a l' => LCons a (LList_decomp_n m l')
end.
```

Unfolding Techniques (3/3)

- We want a function that decomposes n times.

```
Fixpoint LList_decomp_n (A : Set) (n : nat) (l : LList A) :
LList A :=
match n, l with
| O, l => l
| S m, LNil => LNil
| S m, LCons a l' => LCons a (LList_decomp_n m l')
end.
```

- We also need to prove equality:

```
Lemma general_LList_decomposition_lemma :
forall (n : nat) (A : Set) (l : LList A), LList_decomp_n n l = l.
```

List exponentiation (1/5)

- Function `omega` to compute $u^\omega = uuuuu\ldots$ for some lazy list u.

List exponentiation (1/5)

- Function `omega` to compute $u^\omega = uuuuu\ldots$ for some lazy list u.
- First, we will define `general_omega`, which computes uv^ω for lazy lists u and v.

List exponentiation (2/5)

```
CoFixpoint general_omega (A : Set) (u v : LList A) : LList A :=
match v with
| LNil => u
| LCons b v' =>
match u with
| LNil => LCons b (general_omega v' v)
| LCons a u' => LCons a (general_omega u' v)
end
end.
```

List exponentiation (2/5)

```
CoFixpoint general_omega (A : Set) (u v : LList A) : LList A :=
match v with
| LNil => u
| LCons b v' =>
match u with
| LNil => LCons b (general_omega v' v)
| LCons a u' => LCons a (general_omega u' v)
end
end.
```

- `omega` is a special case of `general_omega`:

```
Definition omega (A : Set) (u : LList A) : LList A :=
general_omega u u.
```

List exponentiation (3/5)

```
Lemma omega_definition_equality : forall (A : Set)(v : LList A),
general_omega LNil v = general_omega v v.
```

List exponentiation (4/5)

- LAppend will concatenate two lazy lists.

List exponentiation (4/5)

- LAppend will concatenate two lazy lists.

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

List exponentiation (4/5)

- LAppend will concatenate two lazy lists.

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

```
Lemma LAppend_LNil : forall (A : Set) (v : LList A),
LAppend LNil v = v.
```

List exponentiation (5/5)

```
Lemma omega_unfold : forall (A : Set) (u : LList A),
omega u = LAppend u (omega u).
```

List exponentiation (5/5)

```
Lemma omega_unfold : forall (A : Set) (u : LList A),
omega u = LAppend u (omega u).
```

- This is not provable!

Coinductive Types in Coq

Jan Heemstra and Erik Voogd

Type theory seminar

Radboud University Nijmegen

April 23, 2019