

Infinite Objects and Proofs ¹

Pierre Castéran

Beijing, August 2009

¹This lecture corresponds to the chapter 13 : “Infinite Objects and Proofs” of the book.

In this lecture, we shall see how to represent in *Coq* infinite or potentially infinite data structures. Such objects may appear in computer science (e.g. the sequence of inputs or outputs of some process), or in mathematics (for instance, the infinite sequence of all prime numbers, 2, 3, 5, 7, 11, ...)

Of course, there is no hope to represent **any** infinite data structure in a bounded computer's memory.

However, we shall present some constructs and computation mechanisms which help us to **simulate** the building and **exploration** of such objects.

For instance, building in *Coq* the infinite sequence of prime numbers or the execution trace of some process amounts to create some (**finite**) term and the possibility to access sequentially to any item of this sequence.

We shall learn how to :

- ▶ declare types for this kind of structures,
- ▶ build terms of these types (data and proofs),
- ▶ use these structures (as arguments of functions, in proofs.)

The examples we will study use *potentially infinite lists*, which correspond to, for instance, the output stream of a process which can either stop or run indefinitely.

Definition of a co-inductive type

Set Implicit Arguments.

```
CoInductive LList (A: Type) : Type := (* lazy lists *)
| LNil : LList A
| LCons : A -> LList A -> LList A.
```

Implicit Arguments LNil [A].

The constants **LNil** and **LCons** are the two *constructors* of the type **LList A**.

There is *no* induction principle, which would say that any such list could be obtained by a finite number of constructor applications (and thus would be finite).

It is easy to build finite lists of our new type :

```
Check (LCons 1 (LCons 2 (LCons 3 LNil))).
LCons 1 (LCons 2 (LCons 3 LNil))
  : LList nat
```

But how to build infinite lists ?

```
(* the list of natural numbers starting from n *)
Fixpoint from (n:nat) : LList nat
  := LCons n (from (S n)).
```

Error message; bad formed recursive definition !

Defining functions using CoFixpoint

Infinite objects can be defined by means of non-ending methods of construction, like in lazy functional programming languages. The **CoFixpoint** command can be used in *Coq* for that purpose.

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

```
Definition nat_stream := from 0.
```

```
CoFixpoint repeat (A:Type)(a: A) := LCons a (repeat a).
```

There is also a construct **cofix** for defining anonymous functions for building infinite objects.

CoFixpoint is not so simple

Let's try to do some tests on `from` and `repeat`.


Eval `simpl in (from 2)`.

= from 2
: LList nat

Eval `compute in (from 2)`.

*= (cofix from (n : nat) : LList nat := LCons n (from (S n))) 2*²
: LList nat

A term of the form `cofix f x ... := t` is *not* spontaneously unwinded, (e.g. `from 2` into `LCons 2 (from 3)`), *since it would lead to infinite computations*.

²The orange term is just the value of the constant `from` 

Lazy computing

A term built with a cofixpoint operator is unwinded *only* when destructured by a matching construct, *i.e.* when its contexts wants to inspect the constructor it has been built with.

```
Eval compute in (match from 2 with
  | LNil ⇒ None
  | LCons a l' ⇒ Some a
end).
```

= Some 2
: option nat

The **match construct** surrounding the **co-fixpoint term** asks for its structure (constructor and arguments), and forces it to be unwinded into **LCons 2 (from 3)**.

Examples of programming with lazy lists

```

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

```

```

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

```

Eval compute in (LHead (LTail (LTail (from 3)))).
= Some 5 : option nat

Guard conditions

The interaction between matching constructs and the co-fixpoint unwinding mechanism, and the requirement on termination of computations lead to the following principle :

Any unwinding of some cofix-point term must produce some constructor (reclaimed by some `match`).

A stronger condition is implemented in `Coq` :

Any recursive call of a function defined as a co-fixpoint must be the argument of a non-empty term built only with constructors.

```

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

```

If *Coq* accepted this definition, the evaluation of the following term would lead to a non-ending computation.

```

filter
  (fun i:nat => match i with | 0 => false
                    | _ => true end)
  (repeat 0)

```

A well-formed definition

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

In the next class, we shall study some properties of this function.

```

CoFixpoint LMap(A B:Type)(f : A -> B)(u :LList A)
  : LList B :=
  match u with
  | LNil => LNil
  | LCons a u' => LCons (f a) (LMap f u')
  end.

```

```

Eval compute in (LHead (LTail
                        (LMap (fun i => i * i) (from 3)))).

```

Some 16 : option nat

Inductive predicates on a co-inductive type

There is no problem at all for defining an inductive property on such a type as `LList A`.

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

```
Lemma L123 : Finite (LCons 1 (LCons 2 (LCons 3 LNil))).
```

Proof.

```
  repeat constructor.
```

Qed.

Co-inductive predicates

A co-inductive predicate is defined as an “ordinary” co-inductive type, of sort Prop. For instance, let us define the **Infinite** predicate.

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

Notice that this predicate has a unique constructor; unlike most inductive predicates, there is no base case (non recursive constructor).

The following lemma is easily proved using a case analysis on l , and inversion on hypotheses `Infinite LNil` and `Infinite (LCons a l)`

```
Lemma Tail_of_Infinite : forall (A:Type)(l : LList A),
    Infinite l ->
    Infinite (LTail l).
```

Proof.

```
destruct l.
```

2 subgoals

A : Type

=====

Infinite LNil -> Infinite (LTail LNil)

subgoal 2 is:

Infinite (LCons a l) -> Infinite (LTail (LCons a l))

```
intro H;inversion H. (* inversion 1 *)
```

1 subgoal

A : Type

a : A

l : LList A

=====

Infinite (LCons a l) -> Infinite (LTail (LCons a l))

intro H; inversion_clear H.

1 subgoal

H : Infinite (LCons a l)

H1 : Infinite l

=====

Infinite (LTail (LCons a l))

simpl;assumption.

Qed.

Infinite Proofs

We would like to prove in *Coq* the proposition

$\forall n : \text{nat}, \text{Infinite } (\text{from } n)$.

In other words, we want to build a function that associates to any n a term of the co-inductive type

$\text{Infinite } (\text{from } n)$, *i.e.* an infinite proof tree of the form :

```
Infinite_LCons n (from n)
  (Infinite_LCons (S n) (from (S n)))
    (Infinite_LCons (S (S n)) (from (S (S n))))
      ...
```

Let us look at some problems we have to solve :

If we manage to get the following subgoal in our proof :

1 subgoal

```
Exp2_Infinite : forall n : nat, Infinite (from n)
n : nat
=====
Infinite (LCons n (from (S n)))
```

then this subgoal is solved by the tactic `constructor;apply Exp2_Infinite`.

We will see now how to get this subgoal.

The tactic `cofix`

The tactic `cofix` helps to build proof terms of type $\forall x : A, P x$ of the form `cofix $H (x : A) : P x := t$` , where t is a term of type $\forall x : A, P x$ in the context Γ augmented with the hypothesis $H : \forall x : A, P x$. The term t must be guarded, in order to make the proof term well formed.

The tactic `cofix` proceeds as follows :

1. On a goal of the form $\forall x : A, P x$, where P is a co-inductive type family, the call `cofix H` adds to the current context an hypothesis $H : \forall x : A, P x$,
2. the user tries then to solve the goal $\forall x : A, P x$, building a solution t in which all recursive calls to H are guarded,
3. when the whole proof term is built (after the `Qed` command), the guard condition is checked by the system.

In our example, we would like to build a proof term t for getting a proof term like :

```
cofix H (n:nat) : Infinite (from n) := t :  
∀ n:nat, Infinite (from n)
```

As for functions like `from`, the whole term must be guarded, *i.e.* each recursive call of `H` must be surrounded only by calls of `Infinite`'s constructor `Infinite_LCons`.

The tactic `cofix` at work

Lemma `from_Infinite` : forall n, Infinite (from n).
 Proof.

1 subgoal

=====

forall n : nat, Infinite (from n)

`cofix` H.

1 subgoal

H : forall n : nat, Infinite (from n)

=====

forall n : nat, Infinite (from n)

A bad attempt

1 subgoal

$H : \text{forall } n : \text{nat}, \text{Infinite (from } n)$

=====

$\text{forall } n : \text{nat}, \text{Infinite (from } n)$

assumption.

Proof completed.

Qed.

Error: Recursive definition of H is ill-formed.

In environment

$H : \text{forall } n : \text{nat}, \text{Infinite (from } n)$

Unguarded recursive call in "H".

What happened? The use of **assumption** on the hypothesis **H** led to the building of a non-guarded proof term

```
cofix H (n :nat) : Infinite (from n) := H
```

What we have to do is forcing the use of **Infinite**'s constructor in order to make it surround the recursive call of **H**.

Let us consider some natural number n .

`intro n.`

1 subgoal

$H : \text{forall } n : \text{nat}, \text{Infinite (from } n)$

$n : \text{nat}$

=====

$\text{Infinite (from } n)$

Since the term `from n` doesn't have the form `LCons t1 t2`, we have first to replace the term `from n` by its unwinded form `LCons n (from (S n))`.

The *ad-hoc* tactic `unwind` (available in this class's list of exercises) allows us to introduce an equality in the context :

```
unwind (from n) (LCons n (from (S n))).
```

1 subgoal

H : forall n : nat, Infinite (from n)

n : nat

eg : from n = LCons n (from (S n))

=====

Infinite (from n)

End of the proof

rewrite eg.

1 subgoal

H : forall n : nat, Infinite (from n)

n : nat

eg : from n = LCons n (from (S n))

=====

Infinite (LCons n (from (S n)))

constructor;auto.

Qed.

The proof term built by this proof is guarded, since the subterm that corresponds to an application of **H** is surrounded by constructors.

An Important Definition

We say that two (lazy) lists l and l' are *extensionally* equal if they share the same elements in the same order. This relationship can be defined as a co-inductive predicate :

```
CoInductive LList_eq (A:Type)
  : LList A -> LList A -> Prop :=
| LList_eq_LNil : LList_eq LNil LNil
| List_eq_LCons : forall a l l',
  LList_eq l l' ->
  LList_eq (LCons a l) (LCons a l').
```

We also say that l and l' are *bisimilar*.

In *Coq*, there is no way to prove the following theorem :

$$\forall (A:\text{Type})(l\ l': \text{LList } A), \text{LList_eq } l\ l' \rightarrow l=l'.$$

On the other hand, we can “easily” prove the following equivalence :

```
Lemma LList_eq_LNth : forall A (l l': LList A),  
  LList_eq l l' <->  
  (forall n, LNth n l = LNth n l').
```

A Last example

Let us consider the following statement :

Lemma `Infinite_not_Finite` : $\forall A (l:LList A),$
`Infinite l` $\rightarrow \sim$ `Finite l`.

Proof.

We *cannot* use `cofix` for solving this goal, since `cofix` helps to *prove* statements in which the associated co-inductive type is in the *conclusion* position.

Moreover, no induction on `l` is possible !

On the other hand, it is possible to do an induction on an hypothesis of type **Finite I** :

```
intros A l H H0; generalize H.
```

1 subgoal

A : Type

l : LList A

H : Infinite l

H0 : Finite l

=====

Infinite l → False

```
induction H0.
```


2 subgoals

A : Type

H : Infinite LNil

=====

Infinite LNil → False

subgoal 2 is:

Infinite (LCons a l) → False

`inversion 1. (* solves the first subgoal *)`

1 subgoal

a : A

l : LList A

H : Infinite (LCons a l)

H0 : Finite l

IHFinite : Infinite l → Infinite l → False

=====

Infinite (LCons a l) → False

`inversion_clear 1.`

1 subgoal

...

H0 : Finite I

IHFinite : Infinite I → Infinite I → False

H2 : Infinite I

=====

False

auto.

Qed.

Conclusion

- ▶ Handling infinite objects in a finite computer is one the most exciting activities in computer science,
- ▶ Using co-inductive definitions is rather more difficult than using inductive ones,
- ▶ Don't forget that **injection**, **discriminate**, **inversion** and pattern-matching also work in this context,
- ▶ A good practice (running examples, doing exercises) is necessary for feeling comfortable with co-inductive types.