13

** Infinite Objects and Proofs

Reasoning about infinite objects while staying in the finite world of a computer is one of the most fascinating uses of proof tools.

Inductive proof techniques already make it possible to prove statements for infinite collections of objects, that is, integers, binary trees, and so on. Of course, each of these objects is built in a finite number of steps and this is the intuitive justification for induction. We propose taking a further step, with techniques to build and handle infinite objects, integrated in the *Coq* system by Gimenez [43, 44]. The main example that we use in this chapter consists in streams, which are especially adapted to model reactive systems. In domains such as communication, energy, or transportation, infinite execution is the norm rather than the exception.

13.1 Co-inductive Types

The types we often study are extensions of classical data types, that is, infinite or potentially infinite lists, potentially infinite trees, and so on. Most of our examples deal with finite or infinite lists and some exercises deal with binary trees that may contain an infinity of nodes.

13.1.1 The `CoInductive` Command

To understand the concept of co-inductive types, we can compare it with the concept of inductive types. Terms in an inductive type are obtained by repeated uses of the constructors provided in the definition. Moreover, the terms should be constructed in such a way that there cannot be infinite branches of subterms. This constraint is actually expressed by the induction principle associated with the type. Co-inductive types are similar to inductive types, in the sense that terms should still be obtained by repeated uses of the constructors. However, there is no induction principle and the branches in the inductive types may be infinite.

The command to define a new co-inductive type is thus very similar to the command to define a new inductive type. We have to provide the type name, its own type, and the names and types of its constructors. For this reason, definitions of co-inductive types will be the same as definitions of inductive types, with only one exception: we replace the keyword `Inductive` by the keyword `CoInductive`.

13.1.2 Specific Features of Co-inductive Types

The fact that terms should be obtained through the constructors is ensured by the possibility of defining terms by pattern matching, as with inductive types. Recursive functions cannot be defined in the same manner. Because we want to preserve the property that every function in *Coq* represents a terminating computation, we cannot consider functions that perform the complete traversal of terms in a co-inductive type. However, we can consider a class of *lazy* recursive functions that build infinite terms in co-inductive types. The terms these functions produce may be infinite, but as long as we require only to see a finite part of these terms, these functions only need to perform finite computations. These functions will be described below as *co-recursive* functions. The most characteristic aspect of these functions is that they *build* values in co-inductive types, while the recursive functions we studied in previous chapters *consume* values in inductive types. The term "co-inductive type" stems from this duality: co-inductive types are the co-domains (the ranges) of co-recursive functions while inductive types are the domains of recursive functions.

We shall also see that Leibniz equality (i.e., `eq`) is too strong to be used to compare co-inductive values. Whenever we cannot prove that two objects built with distinct constructions are identical, we will have to content ourselves with a weaker notion of equivalence: two objects are "equal" if their—maybe infinite—exploration always finds the same component in the same place. Let us say that two such objects are *bisimilar*.

The next three sections describe examples of co-inductive types. Pattern matching is described in Sect. 13.2.2 and co-recursive functions are described in Sect. 13.3.

13.1.3 Infinite Lists (Streams)

The `Streams` module from the *Coq* library defines a type operator for infinite sequences called `Streams:Set→Set`. If A is a type in the `Set` sort, the type "`Stream` A" contains infinite sequences of elements of type A. It is defined with a single constructor named `Cons`:

```
CoInductive Stream (A:Set): Set :=
 Cons : A → Stream A → Stream A.
```

An important difference with the definition of lists from Sect. 6.4.1 is that there is no constructor for the empty list. Thus, we cannot construct finite lists and every element has the form "`Cons` a l."

13.1.4 Lazy Lists

The only difference between the type "`LList` A," used to build finite or infinite lists, and the type "`Stream` A" is that, for the type "`LList A`," there exists a constructor `LNil` for empty lists. One can consider a lazy list as the output stream of some process whose behavior can be either finite or infinite. For this reason, let us call a *stream* or *sequence* any inhabitant of the type "`LList A`," since this type is more general than the type "`Stream A`" provided in the *Coq* libraries.

```
Set Implicit Arguments.

CoInductive LList (A:Set) : Set :=
  LNil : LList A
| LCons : A → LList A → LList A.

Implicit Arguments LNil [A].
```

The prefix `L` (for "lazy") given to most of the constants we define is to avoid confusion with the similar constants from the modules `List` and `Stream` in the *Coq* library. From a set-theoretic point of view, we could say that `LList` is the largest set of terms built with constructors `LNil` and `LCons`, and this includes both finite and infinite terms. As for inductive types, constructors are considered injective and distinguishable (two different constructors always return different results). The tactics `injection` (described in Sect. 6.2.3.1) and `discriminate` (Sect. 6.2.2.2) are thus also relevant for co-inductive types.

13.1.5 Lazy Binary Trees

Finite or infinite binary trees (or *lazy* binary trees) labeled with values of type A can be described using the following co-inductive definition:

```
CoInductive LTree (A:Set) : Set :=
  LLeaf : LTree A
| LBin : A → LTree A → LTree A → LTree A.

Implicit Arguments LLeaf [A].
```

The problems we can encounter with these trees are more complex than for lists. A tree in the type "`LTree` A" may be finite or infinite; when it is infinite it may still have some finite branches or it may have only infinite branches.

13.2 Techniques for Co-inductive Types

It may seem paradoxical to say that we build infinite terms, since we work in the bounded framework of a computer. The first problem we need to solve is the representation problem. This situation is similar to the simulation of streams in applicative languages with a call-by-value strategy. Such a simulation relies on anonymous functions (see [18]). In general, a finite representation of an infinite object is suitable if we can use it to obtain every component of the object by a finite computation. For instance, we should be able to use the finite representation of a lazy list l to determine whether this list contains an nth element and to know its value, for every n. We should be able to use the representation of lazy trees to determine whether a given access path leads to a leaf, an internal node, or does not exist in the tree.

13.2.1 Building Finite Objects

Co-inductive types are defined by a collection of constructors. We can apply these constructors a finite number of times to obtain finite objects, provided there exists at least one non-recursive constructor. For instance, we can build finite terms of type "`LList` A" as was done for the lists of Sect. 6.4.1, simply by repetitive uses of the constructor `LCons`, starting from the constant `LNil`. The following example builds a list containing the integers 1, 2, and 3 in this order. Note that thanks to implicit arguments, the type argument to `LCons` is omitted because it can be inferred from the type of the second argument.

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

On the other hand, it is not possible to build a finite object of type `Stream`.

Exercise 13.1 ** Define an injection mapping every list of type "`list` A" to a list of type "`LList` A" and prove that it is injective.*

13.2.2 Pattern Matching

Since infinite objects are represented by a possibly complex encoding, it is important to provide a simple way to obtain their components. We can use the fact that every term of co-inductive type C is necessarily of the form "$c\ t_1 \ldots t_n$" where c is a constructor of C. The `match` construct (introduced in Sect. 6.1.4 for inductive types) is the standard tool to reach the components $t_1 \ldots t_n$. Figure 13.1 gives a few functions for accessing the components of lazy lists. Here is an example using one of these functions:

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

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

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

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

Fixpoint LNth (A:Set)(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.
```

Fig. 13.1. Access functions for lazy lists

Exercise 13.2 ** Define predicates and access functions for the type of lazy binary trees:*

- *`is_LLeaf`: to be a leaf,*
- *`L_root`: the label of the tree root (when it exists),*
- *`L_left_son`,*
- *`L_right_son`,*
- *`L_subtree`: yields the subtree given by a path from the root (when it exists),*
- *`Ltree_label`: yields the label of the subtree given by a path from the root (when it exists).*

The paths are described as lists of directions where a direction is defined as follows:

```
Inductive direction : Set := d0 (* left *) | d1 (* right *).
```

13.3 Building Infinite Objects

In this section, we study how to represent infinite structures in a finite manner. We cannot provide a representation for *all* infinite structures. A simple cardinality argument is enough to convince us that this is not possible. For instance, infinite lists of boolean values can be used to represent non-denumerable sets like the real interval $[0, 1]$ while finite representations can only be used to represent denumerable sets. Still, there are infinite objects that we can describe.

13.3.1 A Failed Attempt

We study the way to build the stream of all natural numbers. Of course, it is impossible to build by hand an infinite term with the following form:

```
LCons 0 (LCons 1 (LCons 2 (LCons 3 ... ))).
```

Another way to proceed is to consider the streams "`from` n" of all numbers starting from n in a symbolic way. We should have the following equality:

`from` n `=` `(LCons` n `(from` $n+1$`))`

We are tempted to use a recursive definition, using `Fixpoint`:

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

However, this is not correct. This definition is not well-formed because n is not structurally decreasing in the recursive call, quite the contrary. It is refused by the *Coq* system:

Error: Recursive definition of ''from'' is ill-formed.
In environment n : nat,
Recursive call to ''from'' has principal argument equal to
''S n'' instead of a subterm of ''n''

13.3.2 The `CoFixpoint` Command

The syntax of the `CoFixpoint` command is close to the syntax of the `Definition` command. However, it makes recursive calls possible and therefore infinite recursion leading to infinite data is possible. Here is how we can define the list of all natural numbers starting at n:

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

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

There is also an anonymous form of `cofixpoint`, called `cofix`, used in the same way as `fix`:

```
Definition Squares_from :=
  let sqr := fun n:nat ⇒ n*n in
  cofix F : nat → LList nat :=
     fun n:nat ⇒ LCons (sqr n)(F (S n)).
```

The `Cofixpoint` command is closer to the `Definition` command because co-recursion relies on the fact that the function's co-domain is a co-inductive type and there is no constraint on the function's domain, while the `Fixpoint` command imposes a constraint on the function's domain and we need to state which input argument is the principal argument. From now on, functions defined with the `cofixpoint` command or the `cofix` construct will be called *co-recursive* functions.

Guard Conditions

Not all recursive definitions are allowed using the `cofixpoint` command. First, the result type must be a co-inductive type, second there is a syntactic condition on recursive calls that is somehow related to the syntactic condition on recursive calls in the `Fixpoint` command. A definition by `cofixpoint` is only accepted if all recursive calls (like "`from (S n)`" in our example) occur inside one of the arguments of a constructor of the co-inductive type. This is called the *guard condition*. This guard condition is inspired by lazy programming languages in which constructors do not evaluate their arguments. This prevents the evaluation of "`from 0`" from looping. In our definition of `from`, the guard condition is satisfied, and the only recursive call "`from (S n)`" occurs as the second argument of the constructor `LCons`.

To motivate this guard condition, let us consider the ways in which infinite objects are used. We can read the data in an infinite object by pattern matching and we would like all computation to terminate, including pattern matching on an infinite object. The guard condition ensures that every co-recursive call produces at least one constructor of the co-inductive type being considered. Thus, a pattern matching operation on data in a co-inductive type requires a finite number of co-recursive calls to decide the branch to follow. Let us consider a few examples:

```
Eval simpl in (isEmpty Nats).
    = False : Prop
```

The `isEmpty` predicate is defined using pattern matching. After $\beta\delta$-reductions, the term to simplify has the following shape:

```
  match (cofix from (n:nat): LList nat :=
       LCons n (from (S n))) 0 with
| LNil ⇒ True
| LCons _ _ ⇒ False
end
    : Prop
```

The pattern matching construct provokes an expansion of the co-fixpoint expression and this produces the `LCons` constructor applied to 0 and the recursive call "`from 1`." The pattern matching clause for this value yields `False`.

On the other hand, simplifying an expression that is not inside a pattern matching construct does not provoke any expansion of co-fixpoint expressions:

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

Eval compute in (from 3).
= (cofix from (n:nat): LList nat :=
       LCons n (from (S n))) 3
   : LList nat
```

Expansions can be iterated when pattern matching occurs in structurally recursive functions for other inductive types. The `LNth` function is structurally recursive over an argument of type `nat`.

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

```
Eval compute in (LNth 19 (from 17)).
```
= Some 36 : option nat

13.3.3 A Few Co-recursive Functions

We advise the reader to check that the guard condition is satisfied in all the examples in this section.

13.3.3.1 Repeating the Same Value

The function `repeat` takes as argument a value and yields a list where this value is repeated indefinitely:

```
CoFixpoint repeat (A:Set)(a:A) : LList A := LCons a (repeat a).
```

13.3.3.2 Concatenating Lists

A solution to concatenating potentially infinite lists is to analyze the first list by pattern matching and to decide what value should be returned depending on the pattern matching cases. This is an example of a function that has a potentially infinite list as *input* and produces another one. Nevertheless, the guard is expressed with respect to the output rather than the input.

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

Here are a few examples combining `LAppend`, `repeat`, and `LNth`. In the first example, the 123 recursive calls to `LNth` provoke as many recursive calls to `LAppend` and to `repeat`. In the second example, the first argument to `LAppend` is exhausted after the first three recursive calls and the second argument is then used.

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

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

13.3.3.3 Repeating the Same Sequence

The last example is more complex. We want to generalize the `repeat` function by considering the infinite iteration of a sequence "u: `LList` A." For instance, the infinite iteration of "`LCons 0 (LCons 1 LNil)`" should return a stream that alternates the values `0` and `1`. When u is infinite, the result is u itself. When u is empty, the result is also the empty stream.

A direct definition by `CoFixpoint` does not work. We would have to construct the infinite iteration of "`Lcons` a v" from the infinite iteration of v, but there is no simple correspondence. To solve this problem, it is better to solve a more general problem. We first consider the problem of concatenating a sequence u with an infinite repetition of another sequence v, a value which we write temporarily as uv^ω.

- If v is empty the result is u.
- Otherwise, consider $v =$ `LCons` b v':
  - if u is empty then the result is a sequence with b as head and v'(`LCons` b v')$^\omega$ as tail,
  - otherwise, consider $u =$ `LCons` a u': the result is a sequence with a as head and $u'v^\omega$ as tail.

The function computing uv^ω can be defined as a co-recursive function. We can apply this function to $u = v$ to solve the initial problem.

```
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.

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

These functions may look quite complex but we see later how to obtain a few simple lemmas that make it easier to reason on them.

Exercise 13.3 *** Build a binary tree containing all strictly positive integers.*

Exercise 13.4 ** Define a function* `graft` *on "*`LTree A`*" so that "*`graft` t t'*" is the result of replacing all leaves of t by t'.*

13.3.4 Badly Formed Definitions

Of course, functions that do not satisfy the guard conditions are rejected. Here are a few classical examples, most of which were described in Gimenez' work [44].

13.3.4.1 Direct Recursive Calls

The following definition of a "filter"—a functional that takes from a stream only those elements that satisfy a boolean predicate—is not accepted, because one of the recursive calls to `filter` appears directly as the value of one of the cases:

```
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.
```

If *Coq* accepted this definition, evaluating the following term would trigger a non-terminating computation:

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

Another example is the following definition, where the first call to the function `buggy_repeat` is not guarded:

```
CoFixpoint buggy_repeat (A:Set)(a:A) : (LList A) :=
   match buggy_repeat a with
     LNil ⇒ LNil
   | LCons b l' ⇒ LCons a (buggy_repeat a)
   end.
```

13.3.4.2 Recursive Calls in a Non-constructor Context

In the following definition, one of the internal calls to `F` is only included in a call of `F` itself, and `F` is not a constructor of the targeted co-inductive type:

```
CoFixpoint F (u:LList nat) : LList nat :=
  match u with
    LNil ⇒ LNil
  | LCons a v ⇒ match  a with
                  0  ⇒ LNil
                | S b ⇒ LCons  b (F (F v))
                end
  end.
```

Determining whether this function always terminates would require an analysis that is too complex to be automated. The *Coq* system relies on a rather simple criterion and rejects this kind of definition.

Exercise 13.5 ** Define the functional with the following type*

```
LMap:prodsymA B:Set, (A→B) → LList A → LList B
```

such that "`LMap f l`*" is the list of images by* f *of all elements of* l*.*
Can you define the functional with the following type

```
LMapcan:∀ A B:Set, (A→(LList B)) → LList A → LList B
```

such that "`LMapcan f l`*" is the concatenation using* `LAppend` *of the images by* f *of all elements of* l*?*

13.4 Unfolding Techniques

We must now study the techniques to reason on co-recursively defined functions. We have seen in the previous section that unfolding a recursive definition is very restricted. We can illustrate this with an attempt to compute the concatenation of two finite sequences:

```
Eval simpl in
  (LAppend (LCons 1 (LCons 2 LNil))(LCons 3 (LCons 4 LNil))).
 = LAppend (LCons 1 (LCons 2 LNil))(LCons 3 (LCons 4 LNil))
   : LList nat
```

No substantial modification is performed and `LAppend` still appears as applied to the two streams it had as arguments, while we would have expected to obtain a single term written only with `LCons`, `LNil`, and natural numbers. The next example shows a proof attempt that fails for the same reason:

```
Theorem LAppend_LCons :
 ∀(A:Set)(a:A)(u v:LList A),
   LAppend (LCons a u) v = LCons a (LAppend u v).
Proof.
 intros; simpl.
 ...
 ============================
   LAppend (LCons a u) v = LCons a (LAppend u v)
```

The `simpl` tactic did not make any progress.

13.4.1 Systematic Decomposition

We saw in Sect. 13.3.3.2 that an access operation provokes the unfolding of a co-recursive function like `LAppend`. In general, if a term t has a co-inductive type C, then there exists a constructor of C so that t is obtained by applying this constructor. This property can be expressed as an equality between t and a pattern matching construct. This equality is the statement of a *decomposition lemma*. Such a lemma can be built for every inductive or co-inductive type, but it is only useful for co-inductive types, because reduction takes care of the decomposition for recursive functions of inductive types (this approach was suggested to us by Christine Paulin-Mohring). For instance, the decomposition lemma for the type of potentially infinite lists is described with an auxiliary function:

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

The following lemma shows that `LList_decompose` really is an identity function on "`list` A." Its proof is a simple case-by-case analysis:

```
Theorem LList_decomposition_lemma :
   ∀(A:Set)(l:LList A), l = LList_decompose l.
Proof.
 intros A l; case l; trivial.
Qed.
```

Exercise 13.6 ** Follow the same approach for the type of potentially infinite binary trees.*

13.4.2 Applying the Decomposition Lemma

From a functional point of view, the function `LList_decompose` is only an identity function on "`list` A," and it seems stupid to define it. From an operational point of view, however, this function is interesting because its application on the result of co-recursive functions provokes an unfolding of these co-recursive functions.

```
Eval simpl in (repeat 33).
    = repeat 33  : LList nat

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

Exercise 13.7 ** Define a function `LList_decomp_n` with type*

```
∀A:Set, nat→ LList A → LList A
```

that iterates the function `LList_decompose`*. For instance, we should have the following dialogue:*

```
Eval simpl in (LList_decomp_n 4
                (LAppend (LCons 1 (LCons 2 LNil))
                         (LCons 3 (LCons 4 LNil)))).
 = LCons 1 (LCons 2 (LCons 3 (LCons 4 LNil)))
    : LList nat

Eval simpl in (LList_decomp_n 6 Nats).
  = LCons 0
      (LCons 1
        (LCons 2
          (LCons 3
            (LCons 4 (LCons 5 (from 6))))))
    : LList nat

Eval simpl in
    (LList_decomp_n 5 (omega (LCons 1 (LCons 2 LNil)))).
   = LCons 1
      (LCons 2
        (LCons 1
          (LCons 2
            (LCons 1
              (general_ omega (LCons 2 LNil)(LCons 1 (LCons 2 LNil))))))))
    : LList nat
```

Generalize the decomposition lemma to this function.

13.4.3 Simplifying a Call to a Co-recursive Function

The decomposition lemma makes it possible to force the expansion of co-recursive calls, when necessary. For instance, we want to prove the equality

```
LAppend Lnil v = v
```

for every type `A` and every list `v` of the type "`LList A`".

Thanks to the decomposition lemma, we can transform this goal into

```
LList_decompose (LAppend LNil v) = v.
```

A case-by-case analysis on `v` leads to the following two goals:

```
LList_decompose (LAppend LNil LNil) = LNil

LList_decompose (LAppend LNil (LCons a v)) = LCons a v.
```

In both cases, simplification leads to a trivial equality. Here is the complete script, with a tactic that simplifies the reasoning process and that will be used extensively throughout the rest of this chapter:

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

Theorem LAppend_LNil : ∀(A:Set)(v:LList A), LAppend LNil v = v.
Proof.
  intros A v.
  LList_unfold (LAppend LNil v).
  case v; simpl; auto.
 Qed.
```

In the same manner, we can prove the lemma `LAppend_LCons` (a proof attempt for this lemma failed at the start of Sect. 13.4).

```
Theorem LAppend_LCons :
  ∀(A:Set)(a:A)(u v:LList A),
    LAppend (LCons a u) v = LCons a (LAppend u v).
Proof.
  intros A a u v.
  LList_unfold (LAppend (LCons a u) v).
  case v; simpl; auto.
Qed.
```

These useful lemmas can be placed in the databases for the tactic `autorewrite`.

```
Hint Rewrite LAppend_LNil LAppend_LCons : llists.
```

Exercise 13.8 *** Prove the unfolding lemmas for the example functions defined in this chapter:*

```
Lemma from_unfold : ∀n:nat, from n = LCons n (from (S n)).

Lemma repeat_unfold :
 ∀(A:Set)(a:A), repeat a = LCons a (repeat a).

Lemma general_omega_LNil : ∀A:Set, omega LNil = LNil (A := A).

Lemma general_omega_LCons :
 ∀(A:Set)(a:A)(u v:LList A),
   general_omega (LCons a u) v = LCons a (general_omega u v).

Lemma general_omega_LNil_LCons :
 ∀(A:Set)(a:A)(u:LList A),
   general_omega LNil (LCons a u) =
```

```
  LCons a (general_omega u (LCons a u)).
```

Conclude with the following lemma:

```
Lemma general_omega_shoots_again : ∀(A:Set)(v:LList A),
  general_omega LNil v = general_omega v v.
```

Remark 13.1 *We would have also liked to give the following lemma as an exercise:*

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

But this cannot be proved. This is not the direct translation of the `omega` *function or its auxiliary function. There is a complex issue when* `u` *is infinite. In fact we can only prove this lemma when* `u` *is finite, but with a much more complex reasoning than for the examples given so far. Another solution that we study in Sect. 13.7 consists in providing an equivalence relation that is weaker than the usual* Coq *equality. Two lists are equivalent if they have the same elements at the same place.*

Exercise 13.9 *** Prove the unfolding lemmas for the function* `graft` *defined in Exercise 13.4.*

13.5 Inductive Predicates over Co-inductive Types

Most of the tools studied in previous chapters for inductive properties still apply for dependent inductive types whose arguments have a co-inductive type. This section does not introduce new notions and only gives a few examples.

A Predicate for Finite Sequences

Since the type "`LList` *A*" contains finite and infinite sequences it is useful to have a predicate `Finite`. A finite sequence is built by a finite number of applications of the constructors `LNil` and `LCons` and it is natural to describe this using an inductive definition:

```
Inductive Finite (A:Set) : LList A → Prop :=
  | Finite_LNil : Finite LNil
  | Finite_LCons :
      ∀(a:A)(l:LList A), Finite l → Finite (LCons a l).

Hint Resolve Finite_LNil Finite_LCons : llists.
```

Constructor application, inversion, and induction can be applied on this inductive predicate without any problem. The following proofs also rely on automatic proofs using `auto` and `autorewrite`:

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

Theorem Finite_of_LCons :
 ∀(A:Set)(a:A)(l:LList A), Finite (LCons a l) → Finite l.
Proof.
 intros A a l H; inversion H; assumption.
Qed.


Theorem LAppend_of_Finite :
 ∀(A:Set)(l l':LList A),
   Finite l → Finite l' → Finite (LAppend l l').
Proof.
  induction 1; autorewrite with llists using auto with llists.
Qed.
```

Exercise 13.10 **** Prove the following lemma that expresses how the function* `omega` *iterates on its argument. Note that this theorem is restricted to finite streams. This is a partial solution to the problem described in Remark 13.1.*

```
Theorem omega_of_Finite :
 ∀(A:Set)(u:LList A), Finite u → omega u = LAppend u (omega u).
```

Hint: use lemmas from Exercise 13.8.

Exercise 13.11 *Define the predicate on "*`LTree` *A" which characterizes finite trees. Prove the equality*

```
graft t (LLeaf A) = t
```

for every finite tree t.

13.6 Co-inductive Predicates

We have seen in Chap. 3 that propositions are types and their proofs are inhabitants of these types. Co-inductive types of sort `Prop` make it possible to

describe co-inductive predicates; proofs of statements using these predicates are infinite proof terms, which we can construct with the `cofix` construct. Nothing distinguishes the construction of co-inductive data from the construction of co-inductive proofs. As an illustrative example, we define the predicate that characterizes infinite lists.

13.6.1 A Predicate for Infinite Sequences

We used an inductive definition to describe the finite sequences of "`LList` *A*": a finiteness proof is a term obtained with a finite number of applications of `Finite_LCons` to an term obtained with `Finite_LNil`. In a symmetric manner, we propose to describe the co-inductive type `Infinite` with only one constructor:

```
CoInductive Infinite (A:Set) : LList A → Prop :=
    Infinite_LCons :
      ∀(a:A)(l:LList A), Infinite l → Infinite (LCons a l).
Hint Resolve Infinite_LCons : llists.
```

With respect to proof techniques, we start by presenting the techniques to prove that a given term is infinite. The techniques to use the fact that a term is infinite are a subset of the techniques seen for inductive types.

13.6.2 Building Infinite Proofs

13.6.2.1 An Intuitive Description

Let us start with a simple example. We want to show that the function `from`, which maps any n to the stream of natural numbers starting from n, yields infinite lists. We should build a term of type "`∀n:nat, Infinite (from n)`." We first present a manual proof, but this is to introduce the more elaborate tools that will be used for other proofs.

A good way to build a term of the required type is to define a co-recursive function in the type "`∀n:nat, Infinite (from n)`." Such a function has to satisfy the guard conditions. In fact, this co-recursive function is a fixpoint of a functional from the type "`∀n:nat, Infinite (from n)`" to itself. We first define this functional, which requires no recursion.

```
Definition F_from :
  (∀n:nat, Infinite (from n))→∀n:nat, Infinite (from n).

 intros H n; rewrite (from_unfold n).
 ...
 H : ∀ n:nat, Infinite (from n)
 n : nat
 ============================
```

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

 split.

  ...
  H : ∀ n:nat, Infinite (from n)
  n : nat
  ============================
  Infinite (from (S n))

 trivial.
Defined.
```

We really took care that the function `H` was only used to provide an argument to a constructor. This corresponds to a guard condition; moreover, we made this function transparent, so that the *Coq* system is able to check the guard condition when reusing this functional in a `cofix` construct:

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

13.6.2.2 The `cofix` Tactic

The elementary steps we have taken in the previous section are automatically taken by the `cofix` tactic. The principle remains the same. There is a hypothesis that cannot be used carelessly. To prove a property P where P is based on a co-inductive predicate, one should construct a term of the form "`cofix` $H:P$ `:=` t" where t has type P in the context with a hypothesis $H:P$ and the term we obtain satisfies the guard condition.

From the user's point of view, the `cofix` H tactic takes charge of introducing the hypothesis H and providing a new goal with statement P. When this goal is solved, the whole proof term is built; then, and only then, is the guard condition verified.

Here is an interactive proof of `from_Infinite` using this tactic:

```
Theorem from_Infinite : ∀n:nat, Infinite (from n).
Proof.
 cofix H.
  ...
  H : ∀ n:nat, Infinite (from n)
  ============================
   ∀ n:nat, Infinite (from n)

 intro n; rewrite (from_unfold n).
 split; auto.
Qed.
```

We suggest that the reader tests this script on a computer, using the `Print` command to check that a co-recursive function is defined.

13.6.3 Guard Condition Violation

In the previous proof, it is the tactic `split` that imposes that the guard condition is satisfied. An attempt to let automatic tactics do the whole job in one shot leads to a proof that is too simple and does not satisfy the guard condition.

In the following script, the tactic "`auto with llists`" prefers a direct use of the hypothesis `H` and the proof term that we obtain is incorrect. This kind of situation is one of the rare cases where the user is mislead in thinking the proof is over because there are no more goals:

```
Lemma from_Infinite_buggy : ∀n:nat, Infinite (from n).
 Proof.
 cofix H.
 auto with llists.
 Proof completed.
 Qed.
 Error: Recursive definition of ''H'' is ill-formed.
In environment
H : ∀ n:nat, Infinite (from n)
unguarded recursive call in ''H''
```

In the case of proofs that can be much more complex than our example, it is sensible to question the perversity of a system that lets the user painfully design a proof term and announces that this term is incorrect only after the complete term is given. Fortunately, an extra command named `Guarded` is provided to test whether the guard condition is respected in the current proof attempt. We advise users to use this command when there is any doubt, especially after each use of an automatic tactic like `assumption` or `auto`, or any explicit use of the hypothesis introduced by the `cofix` tactic.

In our small example, this command makes it possible to detect the problem directly after the automatic tactic has been used:

```
Lemma from_Infinite_saved : ∀n:nat, Infinite (from n).
Proof.
 cofix H.
 auto with llists.
 Guarded.
 Error: Recursive definition of ''H'' is ill-formed.
In environment
H : ∀ n:nat, Infinite (from n)
unguarded recursive call in ''H''
 Undo.
```

```
 intro n; rewrite (from_unfold n).
 split; auto.
 Guarded.
```
The condition holds up to here
```
Qed.
```

Exercise 13.12 ** Prove the following lemmas, using the* `cofix` *tactic:*

```
Lemma repeat_infinite : ∀(A:Set)(a:A), Infinite (repeat a).

Lemma general_omega_infinite :
 ∀(A:Set)(a:A)(u v:LList A),
   Infinite (general_omega v (LCons a u)).
```

Conclude with the following theorem:

```
Theorem omega_infinite :
 ∀(A:Set)(a:A)(l:LList A), Infinite (omega (LCons a l)).
```

Exercise 13.13 *A distracted student confuses keywords and gives an* inductive *definition of being infinite:*

```
Inductive BugInfinite (A:Set) : LList A → Prop :=
    BugInfinite_intro :
      ∀(a:A)(l:LList A),
        BugInfinite l → BugInfinite (LCons a l).
```

Show that this predicate can never be satisfied.

Exercise 13.14 *** Define the predicates "to have at least one infinite branch" and "to have all branches infinite" for potentially infinite binary trees (see Sect. 13.1.5). Consider similar predicates for finite branches. Construct a term that satisfies each of these predicates and prove it. Study the relationships between these predicates; beware that the proposition statement:*

"If a tree has no finite branch, then it contains an infinite branch"

can only be proved using classical logic, in other words with the following added axiom:

```
∀P:Prop, ∼∼P→P.
```

13.6.4 Elimination Techniques

Can we prove theorems where co-inductive properties appear in the premises? Clearly, the induction technique can no longer be used, since lists are potentially infinite. Still, case-by-case analysis and inversion are still available. We illustrate this in a simple example and leave other interesting proofs as exercises.

`LNil` Is Not Infinite

The following proof uses an inversion on "`Infinite (LNil (A:=A))`." Because there is no constructor for `Infinite` concerning the empty list, this inversion proves the theorem immediately.

```
Theorem LNil_not_Infinite :
 ∀A:Set, ~Infinite (LNil (A:=A)).
Proof.
  intros A H; inversion H.
Qed.
```

Exercise 13.15 *** Prove the following statements:*

```
Theorem Infinite_of_LCons :
 ∀(A:Set)(a:A)(u:LList A), Infinite (LCons a u)→ Infinite u.

Lemma LAppend_of_Infinite :
 ∀(A:Set)(u:LList A),
   Infinite u → ∀v:LList A, Infinite (LAppend u v).

Lemma Finite_not_Infinite :
 ∀(A:Set)(l:LList A), Finite l → ~Infinite l.

Lemma Infinite_not_Finite :
 ∀(A:Set)(l:LList A), Infinite l → ~Finite l.

Lemma Not_Finite_Infinite :
 ∀(A:Set)(l:LList A), ~Finite l → Infinite l.
```

Exercise 13.16 *** Prove the following two statement in the framework of classical logic.*[1] *To do these proofs, load the* Classical *module from the* Coq *library.*

```
Lemma Not_Infinite_Finite :
 ∀(A:Set)(l:LList A), ~Infinite l → Finite l.

Lemma Finite_or_Infinite :
 ∀(A:Set)(l:LList A), Finite l ∨ Infinite l.
```

[1] It is impossible to build intuitionistic proofs of these statements. For the first statement, no logical argument can be given to build a proof of "`Finite l`;" of course there is no induction on `l` and a case analysis on `l` makes it possible to conclude only if `l` is empty. For the second one, a strong argument (given by E. Gimenez) expresses that if an intuitionistic proof of this statement existed, then one would be able to conclude that the halting problem is decidable for Turing machines, by considering the lists associated with execution traces.

Exercise 13.17 **** The following definitions are valid in the* Coq *system:*

```
Definition Infinite_ok (A:Set)(X:LList A → Prop) :=
  ∀l:LList A,
    X l →  ∃a:A, (∃l':LList A, l = LCons a l' ∧ X l').
Definition Infinite1 (A:Set)(l:LList A) :=
   ∃X:LList A → Prop, Infinite_ok X ∧ X l.
```

Prove that the predicates `Infinite` *and* `Infinite1` *are logically equivalent.*

13.7 Bisimilarity

This section considers equality proofs between terms of a co-inductive type. We have already proved a few results where the conclusion is such an equality: `LAppend_LNil`, `LAppend_LCons`, `omega_of_Finite`—all have been obtained with a finite sequence of unfoldings. There are examples of equality proofs where finite sequences of unfoldings are not enough. For instance, consider the proof that concatenating any infinite stream to any other stream yields the first stream. Here is a first attempt to perform this proof:

```
Lemma Lappend_of_Infinite_0 :
 ∀(A:Set)(u:LList A),
   Infinite u → ∀v:LList A, u = LAppend u v.
```

The only tool at our disposal is a case analysis on the variable `u`. If we decompose `u` into "`LCons a u'`," we obtain a goal that is similar to the initial goal:

```
  H1 : Infinite u'
  v : LList A
  ============================
   u' = LAppend u' v
```

We see that a finite numbers of these steps will not make it possible to conclude. However, we can restrict our attention to a relation on streams that is weaker than equality but supports co-inductive reasoning.

Exercise 13.18 *Write the proof steps that lead to this situation.*

13.7.1 The `bisimilar` Predicate

The following co-inductive type gives a formal presentation of *finite or infinite proofs* of equalities between streams:

```
CoInductive bisimilar (A:Set) : LList A → LList A → Prop :=
  | bisim0 : bisimilar LNil LNil
  | bisim1 :
      ∀(a:A)(l l':LList A),
        bisimilar l l' → bisimilar (LCons a l)(LCons a l').

Hint Resolve bisim0 bisim1  : llists.
```

A proof of a proposition "`bisimilar` *u v*" can be seen as a finite or infinite proof term built with the constructors `bisim0` and `bisim1`. Of course, these proof terms are usually constructed using the `cofix` tactic with the constraint of respecting the guard condition.

Exercise 13.19 *After loading the module* ***Relations*** *from the* Coq *library, show that* ***bisimilar*** *is an equivalence relation. Among other results, reflexivity shows that the* ***bisimilar*** *relation accepts more pairs of streams than equality.*

Exercise 13.20 *** For a better understanding of the* ***bisimilar*** *relation, we can use the function* ***LNth*** *defined in Fig. 13.1. Show the following two theorems, which establish a relation between* ***bisimilar*** *and* ***LNth****:*

```
Lemma bisimilar_LNth :
 ∀(A:Set)(n:nat)(u v:LList A),
   bisimilar u v → LNth n u = LNth n v.

Lemma LNth_bisimilar :
 ∀(A:Set)(u v:LList A),
   (∀n:nat, LNth n u = LNth n v)→ bisimilar u v.
```

Exercise 13.21 *Prove the following two theorems (the proof techniques are interestingly different):*

```
Theorem bisimilar_of_Finite_is_Finite :
 ∀(A:Set)(l:LList A),
   Finite l → ∀l':LList A, bisimilar l l' → Finite l'.


Theorem bisimilar_of_Infinite_is_Infinite :
 ∀(A:Set)(l:LList A),
   Infinite l → ∀l':LList A, bisimilar l l' → Infinite l'.
```

Exercise 13.22 *Prove that restricting* ***bisimilar*** *to finite lists gives regular equality, in other words*

```
Theorem bisimilar_of_Finite_is_eq :
 ∀(A:Set)(l:LList A),
   Finite l → ∀l':LList A, bisimilar l l' → l = l'.
```

Exercise 13.23 *** Redo the previous exercises for lazy binary trees (see Sect. 13.1.5). Define the relationship* `LTree_bisimilar` *and establish its relation with a function accessing the nodes of a tree, in a similar manner as to what is done in Exercise 13.20.*

13.7.2 Using Bisimilarity

This section shows that the equivalence relation `bisimilar` can be used to express and prove some properties, which were unprovable when using the regular equality.

`LAppend` Is Associative

The associativity of `LAppend`, when expressed using `bisimilar`, is proved by co-induction with a case-by-case analysis for the first argument:

```
Theorem LAppend_assoc :
 ∀(A:Set)(u v w:LList A),
   bisimilar (LAppend u (LAppend v w))(LAppend (LAppend u v) w).

Proof.
 intro A; cofix H.
 destruct u; intros;
 autorewrite with llists using auto with llists.
 apply bisimilar_refl.
Qed.
```

Exercise 13.24 ** Prove that every infinite sequence is left-absorbing for concatenation:*

```
Lemma LAppend_of_Infinite_bisim :
  ∀(A:Set)(u:LList A),
    Infinite u → ∀v:LList A, bisimilar u (LAppend u v).
```

Exercise 13.25 **** Prove that the sequence "*`omega` *u" is a fixpoint for concatenation (with respect to bisimilarity.)*

```
Lemma omega_lappend :
 ∀(A:Set)(u:LList A),
    bisimilar (omega u)(LAppend u (omega u)).
```

Hint: first prove a lemma about `general_omega`.

Exercise 13.26 *** As a continuation of Exercise 13.23, show that a tree where all branches are infinite is left-absorbing for the* `graft` *operation defined in Exercise 13.4.*

13.8 The Park Principle

We adapt to lazy lists a presentation provided in Gimenez's tutorial [43]. A *bisimulation* is a binary relation R defined on "`LList A`" so that when "`R` u v" holds, then either both u and v are empty, or there exist an a and two lists u_1 and v_1 so that u is "`LCons` a u_1," v is "`LCons` a v_1," and the proposition '`R` u_1 v_1" holds. Here is a definition of the predicate that characterizes bisimulations:

```
Definition bisimulation (A:Set)(R:LList A → LList A → Prop) :=
  ∀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.
```

Exercise 13.27 **** Prove the following theorem (Park principle):*

```
Theorem park_principle :
 ∀(A:Set)(R:LList A → LList A → Prop),
   bisimulation R → ∀l1 l2:LList A, R l1 l2 →
   bisimilar l1 l2.
```

Exercise 13.28 ** Use the Park principle to prove that the following two streams are bisimilar:*

```
CoFixpoint alter : LList bool := LCons true (LCons false alter).

Definition alter2 : LList bool :=
  omega (LCons true (LCons false LNil)).
```

Hint: consider the following binary relation and prove that it is a bisimulation:

```
Definition R (l1 l2:LList bool) : Prop :=
  l1 = alter ∧ l2 = alter2 ∨
  l1 = LCons false alter ∧ l2 = LCons false alter2.
```

13.9 LTL

This section proposes a formalization of linear temporal logic, LTL [74]. The definitions we present are an adaptation of the work done with D. Rouillard using *Isabelle/HOL* [21]. The work of Coupet-Grimal [28, 29], which is available in the *Coq* contributions, formalizes a notion of the LTL formula restricted to infinite executions (while we consider both finite and infinite executions). A distinction between the two formalizations is that Coupet-Grimal's presentation concentrates on the notion of LTL formulas and their abstract properties, while our presentation concentrates on execution traces and their properties. Still, both contributions use co-induction in a similar manner and we encourage readers to consult both developments.

We start our development by declaring a type A:`Set` and a few variables that are later used for our examples:

```
Section LTL.
Variables (A : Set)(a b c : A).
```

We are interested in properties of streams on A. To make our presentation more intuitive we introduce a notation "`satisfies` l P" for "P l":

```
Definition satisfies (l:LList A)(P:LList A → Prop) : Prop :=
  P l.
Hint Unfold satisfies : llists.
```

We can now define a collection of predicates over "`llist` A."

The `Atomic` Predicate

We can convert any predicate on A into a predicate on "`llist` A." A stream satisfies the predicate "`Atomic` At" if its first element satisfies At:

```
Inductive Atomic (At:A→Prop) : LList A → Prop :=
  Atomic_intro :
    ∀(a:A)(l:LList A), At a → Atomic At (LCons a l).

Hint Resolve Atomic_intro : llists.
```

The `Next` Predicate

The predicate "`Next` P" characterizes all sequences whose tail satisfies P.

```
Inductive Next (P:LList A → Prop) : LList A → Prop :=
  Next_intro : ∀(a:A)(l:LList A), P l → Next P (LCons a l).

Hint Resolve Next_intro : llists.
```

For instance, we show that the stream starting with `a` and followed by an infinity of `b` satisfies the formula "`Next (Atomic (eq b))`":

```
Theorem Next_example :
 satisfies (LCons a (repeat b))(Next (Atomic (eq b))).
Proof.
  rewrite (repeat_unfold b); auto with llists.
Qed.
```

The `Eventually` Predicate

The predicate "`Eventually` P" characterizes the streams with at least one (non-empty) suffix satisfying P. Note that the first constructor is written in such a way that empty streams are excluded.

```
Inductive Eventually (P:LList A → Prop) : LList A → Prop :=
  Eventually_here :
   ∀(a:A)(l:LList A), P (LCons a l)→
                      Eventually P (LCons a l)
| Eventually_further :
   ∀(a:A)(l:LList A), Eventually P l →
                      Eventually P (LCons a l).

Hint Resolve Eventually_here Eventually_further.
```

Exercise 13.29 (**) *Here is a lemma and its proof:*

```
Theorem Eventually_of_LAppend :
 ∀(P:LList A → Prop)(u v:LList A),
 Finite u → satisfies v (Eventually P)→
 satisfies (LAppend u v)(Eventually P).
Proof.
 unfold satisfies; induction 1; intros;
  autorewrite with llists using auto with llists.
Qed.
```

What is the role of finiteness? Is it really necessary? If it is, build a counter-example.

The `Always` Predicate

The predicate "`Always` P" characterizes the streams such that all the suffixes are non-empty and satisfy P. It is natural to use a co-inductive definition with only one constructor:

```
CoInductive Always (P:LList A → Prop) : LList A → Prop :=
  Always_LCons :
  ∀(a:A)(l:LList A),
  P (LCons a l)→ Always P l → Always P (LCons a l).
```

Exercise 13.30 *Prove that every stream satisfying "*`Always` P*" is infinite.*

Exercise 13.31 ** Prove that every suffix of the stream "*`repeat a`*" starts with* `a`*:*

```
Lemma always_a : satisfies (repeat a)(Always (Atomic (eq a))).
```

The F^∞ Predicate

The predicate "`F_infinite` P" characterizes the streams such that an infinity of suffixes satisfy P; this predicate is easily defined with `Always` and `Eventually`.

```
Definition F_Infinite (P:LList A → Prop) : LList A → Prop :=
 Always (Eventually P).
```

Exercise 13.32 ***Show that the infinite sequence w_ω where* `a` *and* `b` *alternate contains an infinity of occurrences of* `a`*.*

The G^∞ Predicate

The predicate "`G_infinite` P" characterizes the streams such that all suffixes except a finite number satisfy P.

```
Definition G_Infinite (P:LList A → Prop) : LList A → Prop :=
 Eventually (Always P).
```

Exercise 13.33 ** Show the following theorems:*

```
Lemma LAppend_G_Infinite :
 ∀(P:LList A → Prop)(u v:LList A),
   Finite u → satisfies v (G_Infinite P)→
   satisfies (LAppend u v) (G_Infinite P).
```

```
Lemma LAppend_G_Infinite_R :
 ∀(P:LList A → Prop)(u v:LList A),
 Finite u → satisfies (LAppend u v) (G_Infinite P)→
 satisfies v (G_Infinite P).

End LTL.
```

13.10 A Case Study: Transition Systems

In this section, we describe the structure of a development on transition systems, that is, automata. Only the statements of theorems are given, the reader can complete the proofs or read the solution on the book's Internet site.[2]

13.10.1 Automata and Traces

An *automaton* is defined using a type to represent *states*, a type to represent *actions*, an *initial* state, and a set of *transitions*, where each transition is described by a source state, an action, and a target state. We choose to represent the set of transitions with a boolean-valued function. The record structure proposed by *Coq* is well-suited to represent automata. Here the record must be defined in the `Type` sort because it contains fields of type `Set` (see Sect. 14.1.2.3).

```
 Record automaton : Type :=
   mk_auto {
    states : Set;
    actions : Set;
    initial : states;
    transitions : states → actions → list states
}.
```

A *trace* is a sequence of actions corresponding to a sequence of transitions of an automaton. Traces can be finite or infinite: when they are finite, they have a *final* state, a deadlock. We give a co-inductive definition of a predicate `Traces`, so that "`Traces` *A q l*" means *l is the execution trace in A from the state q*. The predicate `deadlock` is defined so that "`deadlock` *q*" means *there is no transition leaving from q*.

```
Definition deadlock (A:automaton)(q:states A) :=
  ∀a:actions A, @transitions A q a = nil.

Unset Implicit Arguments.

CoInductive Trace (A:automaton) :
```

[2] `http://www.labri.fr/Perso/~casteran/CoqArt/co-inductifs/index.html`

```
    states A → LList (actions A)→ Prop :=
  empty_trace :
   ∀q:states A, deadlock A q → Trace A q LNil
| lcons_trace :
   ∀(q q':states A)(a:actions A)(l:LList (actions A)),
    In q' (transitions A q a)→ Trace  A  q' l →
    Trace A  q (LCons a l).

Set Implicit Arguments.
```

Exercise 13.34 **** We consider the following automaton:*

```
(* states *)
Inductive st : Set := q0 | q1 | q2.

(* actions *)
Inductive acts : Set := a | b.

(* transitions *)
Definition trans (q:st)(x:acts) : list st :=
  match q, x with
  | q0, a ⇒ cons q1 nil
  | q0, b ⇒ cons q1 nil
  | q1, a ⇒ cons q2 nil
  | q2, b ⇒ cons q2 (cons q1 nil)
  | _, _ ⇒ nil (A:=_)
  end.

Definition A1 :=  mk_auto q0 trans.
```

Draw this automaton, then show that every trace for A_1 contains an infinite number of `b` *actions:*

```
Theorem Infinite_bs :
 ∀(q:st)(t:LList acts), Trace A1 q t →
 satisfies t (F_Infinite (Atomic (eq b))).
```

13.11 Conclusion

A good source on co-inductive types is the work of Gimenez [44, 43] and the documentation of *Coq*. Co-inductive types have also been used to verify the correctness of circuit designs [30].