Introduction to Co-Induction in Coq

Yves Bertot

August 2005

Motivation

- Reason about infinite data-structures,
- Reason about lazy computation strategies,
- Reason about infinite processes, abstracting away from dates.
    - Finite state automata,
    - Temporal logic,
    - Computation on streams of data.

Inductive types as least fixpoint types

- Inductive types are fixpoints of "abstract functions",
  - If $\{c_i\}_{i\in\{1,\ldots,j\}}$ are the constructors of I and $c_i\ a_1\ \cdots\ a_k$ is well-typed then $c_i\ a_1\ \cdots\ a_k \in I$
  - Fixpoint property also gives pattern-matching: if $c_i : T_{i,1}\ \cdots\ T_{i,k} \rightarrow I$ and $f_i : T_{i,1}\ \cdots\ T_{i,k} \rightarrow B$, then there exists a single function $\phi : I \rightarrow B$ such that $\phi(c_i\ a_1 \ldots\ a_k) = f_i\ a_1\ \cdots\ a_k$.
- Initiality:
  - if f_i are functions with type $f_i : T_{i,1}[A/I]\ \cdots\ T_{i,k}[A/I] \rightarrow A$, then there exists a single function $\phi : I \rightarrow A$ such that $\phi(c_1\ a_1\ \cdots\ a_k) = f_i\ a'_1\ \cdots\ a'_k$, where $a'_m = \phi(a_m)$ if $T_m = I$ and $a'_m = a_m$ otherwise.
  - Initiality gives structural recursion.

CoInductive types

- Consider a type C with the first two fixpoint properties,
  - Images of constructors are in C (the co-inductive type),
  - Functions on C can be defined by pattern-matching,
- Take a closer look at pattern-matching:
  - With pattern matching you can define a function $\sigma : C \rightarrow (T_{11} * \ \cdots * \ T_{1k_1}) + (T_{21} * \ \cdots \ * T_{2k_2}) + \cdots$ so that $\sigma(t) = (a_1, \ldots a_{k_i}) \in (T_{i1} * \cdots \ T_{ik_i})$ when $t = c_i \ a_1 \ \cdots \ a_k$
- Replace *initiality* with *co-initiality*, i.e.,
  - If $f : A \rightarrow (T_{11} * \ \cdots * \ T_{1k_1})[A/C] + (T_{21} * \ \cdots \ * T_{2k_2})[A/C] + \cdots$, then there exists a single $\phi : A \rightarrow C$ such that $\phi(a) = c_i \ a'_1 \ \cdots \ a'_{k_i}$ when $f(a) = (T_{i1} * \cdots * \ T_{ik_i})[A/C]$ and $a'_j = \phi(a_j)$ if $T_{ij} = C$ and $a'_j = a_j$ otherwise.

Practical reading of theory

- For both kinds of types,
  - constructors and pattern-matching can be used in a similar way,
- For inductive types,
  - Recursion is only used to consume elements of the type,
  - Arguments of recursive calls can only be sub-components of constructors,
- For co-inductive types,
  - Co-recursion is only used to produce elements of the type,
  - Co-recursive calls can only produce sub-components of constructors.

Theory on an example

- Consider the two definitions:

```
Inductive list (A:Set) : Set :=
  nil : list A | cons : A -> list A -> list A.
CoInductive Llist (A:Set) : Set :=
    Lnil : Llist A
  | Lcons : A -> Llist A -> Llist A.
Implicit Arguments Lcons.
```

- given values and functions `v:B` and `f:A->B->B`, we can define a function `phi : list A -> B` by the following

```
Fixpoint phi (l:list A) : B :=
  match l with
    nil => v | const a t => f a (phi t)
  end.
```

Theory on an example (continued)

- The "natural result type" of pattern-matching on inductive lists is: `unit+(A*list A)`

```
Definition sigma1(A:Set)(l:list A):unit+(A*list A):=
  match l with
    nil => inl (B:=A*list A) tt
  | cons a tl => inr (A:=unit) (a,tl)
  end.
```

- The natural result type of pattern matching on co-inductive lists (type `Llist`) is similar: `unit+(A*Llist A)`
- We can define a co-recursive function `phi : B -> Llist A` if we are able to inhabit the type `B -> unit+(A*B)`.

Categorical terminology

- In the category **Set**, collections of constructors define a functor F,
- for a given object A, $F(A)$ corresponds to the natural result type for pattern-matching as described in the previous slide,
- An F-algebra is an object with a morphism $F(A) \rightarrow A$,
- F-algebras form a category, and the inductive type is an initial object in this category,
- An F-coalgebra is an object with a morphism $A \rightarrow F(A)$,
- F-coalgebras form a category, and the coinductive type is a final object in this category.

Co-Inductive types in Coq

- Syntactic form of definitions is similar to inductive types (given a few frames before),
- pattern-matching with the same syntax as for inductive types.
- Elements of the co-inductive type can be obtained by:
  - Using the constructors,
  - Using the pattern-matching construct,
  - Using co-recursion.

Constructing co-inductive elements

```
Definition ll123 :=
    Lcons 1 (Lcons 2 (Lcons 3 (Lnil nat))).
Fixpoint list_to_llist (A:Set) (l:list A)
    {struct l} :  Llist A :=
 match l with
   nil => Lnil A
 | a::tl => Lcons a (list_to_llist A tl)
 end.
Definition ll123' := list_to_llist nat (1::2::3::nil).
```

- `list_to_llist` uses plain structural recursion on lists and plain calls to constructors.

Infinite elements

- `list_to_llist` shows that `list A` is isomorphic to a subset of `Llist A`
- Lists in `list A` are finite, recursive traversal on them terminates,
- There are infinite elements:
  `CoFixpoint lones : Llist nat := Lcons 1 lones.`
- `lones` is the value of the co-recursive function defined by the *finality* statement for the following `f`:

```
Definition f : unit -> unit+(nat*unit) :=
    fun _ => inr unit (1,tt).
```

Infinite elements (continued)

- Here is a definition of what is called the *finality* statement in this lecture:

```
CoFixpoint Llist_finality
   (A:Set)(B:Set)(f:B->unit+(A*B)):B->Llist A:=
fun b:B => match f b with
  inl tt => Lnil A
| inr (a,b2) => Lcons a (Llist_finality A B f b2)
end.
```

- The *finality* statement is never used in Coq.
- Instead syntactic check on recursive definitions (guarded-by-constructors criterion).

Streams

```
CoInductive stream (A:Set) :  Set :=
  Cons :  A -> stream A -> stream A.
Implicit Arguments Cons.
```

- an example of type where no element could be built without co-recursion.

  ```
  CoFixpoint nums (n:nat) :  stream nat :=
    Cons n (nums (n+1)).
  ```

Computing with co-recursive values

- Unleashed unfolding of co-recursive definitions would lead to infinite reduction,
- A redex appears only when patern-matching is applied on a co-recursive value.
- Unfolding is performed (only) as needed.

Proving properties of co-recursive values

```
Definition Llist_decompose (A:Set)(l:Llist A) : Llist
A :=
  match l with Lnil => Lnil A | Lcons a tl => Lcons a
tl end.
Implicit Arguments Llist_decompose.
```

- Proofs by pattern-matching as in inductive types.

```
Theorem Llist_dec_thm :
   forall (A:Set)(l:Llist A), l = Llist_decompose l.
Proof.
 intros A l; case l; simpl; trivial.
Qed.
```

Unfolding techniques

- The theorem `Llist_dec_thm` is not just an example,
- A tool to force co-recursive functions to unfold.
- Create a redex that maybe reduced by unfolding recursion.

```
Theorem lones_dec :  Lcons 1 lones = lones.
 simpl.
```

============================

Lcons 1 lones = lones

```
pattern lones at 2; rewrite (Llist_dec_thm nat lones);
simpl.
```

============================

Lcons 1 lones = Lcons 1 lones

Proving equality

- Usual equality is an “inductive concept” with no recursion,
- Co-recursion can only provide new values in co-recursive types,
- Need a co-recursive notion of equality.
- Express that two terms are “equal” when then cannot be distinguished by any amount of pattern-matching,
- specific notion of equality for each co-inductive type.

Co-inductive equality

```
CoInductive bisimilar (A:Set) :  Llist A -> Llist A
-> Prop :=
  bisim0 :  bisimilar A (Lnil A)(Lnil A)
| bisim1 :  forall x t1 t2, bisimilar A t1 t2 ->
               bisimilar A (Lcons x t1) (Lcons x t2).
```

Proofs by Co-induction

- Use a tactic `cofix` to introduce a co-recursive value,
- Adds a new hypothesis in the context with the same type as the goal,
- The new hypothesis can only be used to fill a constructor's sub-component,
- Non-typed criterion, the correctness is checked using a `Guarded` command.

Example material

```
CoFixpoint lmap (A B:Set)(f:A -> B)(l:Llist A) :
Llist B :=
  match l with
    Lnil => Lnil B
  | Lcons a tl => Lcons (f a) (lmap A B f tl)
  end.
```

Example proof by co-induction

```
Theorem lmap_bi' :  forall (A:Set)(l:Llist A),
  bisimilar A (lmap A A (fun x => x) l) l.
cofix.
```

1 subgoal

lmap_bi' : forall (A : Set) (l : Llist A),
bisimilar A (lmap A A (fun x : A ⇒ x) l) l

============================

forall (A : Set) (l : Llist A),
bisimilar A (lmap A A (fun x : A ⇒ x) l) l

Example proof by co-induction (continued)

```
intros A l; rewrite
    (Llist_dec_thm _ (lmap A A (fun x=>x) l)); simpl.
...
```

$$
\begin{array}{l}
\rule{0pt}{0pt}\text{============================}\\
\quad \mathit{bisimilar}\ A\\
\quad\quad \mathit{match}\\
\quad\quad\quad \mathit{match}\ l\ \mathit{with}\\
\quad\quad\quad \mid \mathit{Lcons}\ a\ tl \Rightarrow \mathit{Lcons}\ a\ (\mathit{lmap}\ A\ A\ (\mathit{fun}\ x : A \Rightarrow x)\ tl)\\
\quad\quad\quad \mid \mathit{Lnil} \Rightarrow \mathit{Lnil}\ A\\
\quad\quad\quad \mathit{end}\\
\quad\quad \mathit{with}\\
\quad\quad \mid \mathit{Lcons}\ a\ tl \Rightarrow \mathit{Lcons}\ a\ tl\\
\quad\quad \mid \mathit{Lnil} \Rightarrow \mathit{Lnil}\ A\\
\quad\quad \mathit{end}\ l
\end{array}
$$

Example proof by co-induction (continued)

```
case l.
...
```

============================

forall (a : A) (l0 : Llist A),
bisimilar A (Lcons a (lmap A A (fun x : A => x) l0)) (Lcons a l0)

subgoal 2 is:
bisimilar A (Lnil A) (Lnil A)

Example proof by co-induction (continued)

```
intros a k; apply bisim1.
...
```

lmap_bi' : forall (A : Set) (l : Llist A),
bisimilar A (lmap A A (fun x : A => x) l) l

```
...
```

============================

bisimilar A (lmap A A (fun x : A => x) k) k

- A constructor was used, the recursive hypothesis can be used.

```
apply lmap_bi'.
apply bisim0.
Qed.
```

Minimal real arithmetics

- Represent the real numbers in [0,1] as infinite sequences of bits,
- add a third bit to make computation practical.

Redundant floating-point representations

- In usual represenation $1/2$ is both $0.01111\ldots$ and $0.1000\ldots$,
- Every number $p/2^n$ where p and n are integers has two representations,
- Other numbers have only one,
- A number whose prefix is $0.1010\ldots$ (but finite) is a number that can be bigger or smaller than $1/3$,
- When computing $1/3 + 1/6$ we can never decide what should be the first bit of the result.
- Problem solved by adding a third bit : Now `L`, `C`, or `R`.

Explaining redundancy

- A number of the form `L...` is in [0,1/2], (like a number of the form 0.0 . . .),
  - A number of the form `R...` is in [1/2,1], (like a number of the form 0.1 . . .),
  - A number of the form `C...` is in [1/4,3/4].
- Taking an infinite stream of bits and adding a `L` in front divides by 2,
  - Adding a `R` divides by 2 and adds 1/2,
  - Adding a `C` divides by 2 and adds 1/4.

Coq encoding

```
Inductive idigit :  Set := L | C | R.

CoInductive represents :  stream idigit ->
Rdefinitions.R -> Prop :=
  reprL : forall s r, represents s r ->
              (0 <= r <= 1)%R ->
              represents (Cons L s) (r/2)
| reprR : forall s r, represents s r ->
              (0 <= r <= 1)%R ->
              represents (Cons R s) ((r+1)/2)
| reprC : forall s r, represents s r ->
              (0 <= r <= 1)%R ->
              represents (Cons C s) ((2*r+1)/4).
```

Encoding rational numbers

```
CoFixpoint rat_to_stream (a b:Z) : stream idigit :=
  if Z_le_gt_dec (2*a) b then
    Cons L (rat_to_stream (2*a) b)
  else
    Cons R (rat_to_stream (2*a-b) b).
```

Affine combination of redundant digit streams

- compute the representation of

$$\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'},$$

  where x and y are real numbers in [0,1] given by redundant digit streams, and $a \cdots c'$ are positive integers (non-zero when relevant).
- if $2c > c'$ then the result has the form $\mathtt{R}z$ where z is

$$\frac{2a}{a'}x + \frac{2b}{b'}y + \frac{2c - c'}{c'}$$

.

Computation of other digits

- Similar sufficient condition to decide on $\mathtt{C}z$ and $\mathtt{L}z$, for suitable values of z:
  - $$\frac{a}{a'} + \frac{b}{b'} + \frac{c}{c'} \leq \frac{1}{2} \text{ produce } \mathtt{L}$$
  - $$\frac{c}{c'} \geq \frac{1}{4} \text{and} \frac{a}{a'} + \frac{b}{b'} + \frac{c}{c'} \leq 3/4 \text{ produce } \mathtt{C}$$
- if $\frac{a}{a'} + \frac{b}{b'}$ is small enough, you can produce a digit,
- But sometimes necessary to observe x and y.

Consuming input

- if x and y are $\mathtt{L}x'$ and $\mathtt{L}y'$, then
$$\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'}$$
is also
$$\frac{a}{2a'}x' + \frac{b}{2b'}y' + \frac{c}{c'}$$
- Condition for outputting a digit may still not be ensured, but
$$\frac{a}{2a'} + \frac{b}{2b'} = \frac{1}{2}(\frac{a}{a'} + \frac{b}{b'})$$
- Similar for other possible forms of x and y.

Coq encoding

- Use a well-founded recursive function to consume from x and y until the condition is ensured to produce a digit,
- Produce a digit and perform a co-recursive call,
- This style of decomposition between well-founded part and co-recursive is quite powerful (not documented in Coq'Art, though).