Towards general coinduction

Yves Bertot

November 2010

Introduction

- A recapitulation on guarded co-recursion
  - What would be general co-recursion?
- Mixing co-recursion with well-founded induction
- Mixing co-recursion with “size-preserving” higher-order polymorphism

Inroducing co-inductive types

- Introduce the constructors as in inductive types
- Authorize co-recursive definitions
  - Only to produce elements in co-inductive types
  - Authorize only recursive calls in precise locations under constructors, no other function application
- No check on the arguments
- co-recursive functions applied to arguments are not redexes
- Redexes are observations of co-recursive values

An example

```
sieve (p:rs) = p:sieve [r | r <- rs, r 'rem' p /= 0]
primes = sieve [2 ..]
```

Simple Haskell program, manipulating lists

- All lists in this program are infinite
- There is a lot to prove about this program!

Example in Coq

```
Require Import Stream List Bool Arith ZArith.
Open Scope Z_scope.

CoInductive Stream (A : Type) : Type :=
  Cons (x : Z)(tl : stream).
Infix "::" := Cons.
Notation "stream" := (Stream Z).

CoFixpoint ns n := n::ns (n+1).

Fixpoint take n (s : stream) :=
  match n with O => nil
  | S p => let (a,x') := s in (a::take p x')%list
  end.
```

Filtering

```
CoFixpoint filter (p : Z -> bool) (s : stream) :=
  let (x, s') := s in
  if p x then x::filter p s' else filter p s'.
```

- This is rejected
- Every recursive call of `filter` should be under a constructor
  - It should *produce*
- `filter` is a *partial* function

Refreshing the old heavy solution

- Characterize the domain on which filter works
- Explicit that filtering works on this domain
- Fix the relation between successive elements
  - Use a *co-inductive* predicate
- Fix the predicate
- Express that the relation eventually imposes success for the predicate
  - Use well-foundedness

Details

```
CoInductive ct (R : Z -> Z -> Prop) : stream -> Prop :=
  Cct : forall x y s, R x y -> ct R (y::s) ->
    ct R (x::y::s).

Section filter.
Variable R : Z -> Z -> Prop.
Variable p: Z -> Prop.

Hypothesis wf :
  well_founded (fun y x, R x y /\ p x = false).
```

Explaining the details

- The predicate `ct` expresses that a given relation holds between successive elements in a stream
- Filter works if hypothesis `wf` is satisfied
- Much simpler than in Bertot2005

More details

```
Definition sr (s' s:stream) : Prop :=
  R (hd s) (hd s') /\  p (hd s) = false.

Lemma swf : well_founded sr.  Proof. ... Qed.

Definition dec_ct (s : stream) (h : ct R s) :
   {x : Z & {s' | (p x = false -> sr s' s) /\ ct R s'}}.
...
Defined.
```

- sr and swf lift the "well-foundedness" statement to streams
- dec_ct does pattern matching and information for recursion

Horror movie

```
Definition filterb :
    forall s (h : ct R s), Z * {s' | ct R s'} :=
  Fix swf (fun s => ct R s -> Z * {s' | ct R s'})
    (fun s fb h =>
     let (x, (s', (h1, h2))) := dec_ct s h in
    match sumbool_of_bool (p x) with
       left hp => (x, exist (fun s' => ct R s') s' h2)
     | right hp => fb s' (h1 hp) h2
     end).
```

- The use of `fb` stands for the recursive call
- Returns the first satisfying element and the rest of stream

Wrapup

```
CoFixpoint filter (s : stream) (h : ct R s) : stream :=
  let (x, (s', q)) := filterb s h in x::filter s' q.
```

- Note that `p` and `R` are fixed for the last 5 slides
- Then prove that the result satisfies the right properties.

Connectedness

```
Lemma filter_ct :
 forall R1 R2 : Z -> Z -> Prop,
 (forall x y z, R1 x y -> p y = false ->
                                    R2 y z -> R2 x z) ->
 (forall x y, p y = true -> R1 x y -> R2 x y) ->
 forall s (h : ct R s) x,
   R1 x (hd s) -> ct R1 s -> ct R2 (x::filter s h).
```

- Need an extension of `Fix_eq` (Bertot&Balaa2000, Paulin-Mohring)
  - Support for partial functions
  - Also expressed as functions of several arguments
- An "induction theorem" for the filter function

A general approach to well-founded co-induction

- That the input is a stream does not play a role
- Generalization in Bertot&Komendantskaya08
  - Also for trees

Lighter version

```
Fixpoint filter1 (p:Z -> bool)(s:stream)(m:nat) :=
  let (a,tl) := s in if p a then (a,tl) else
  match m with O => (a,tl) | S m' => filter1 p tl m' end.

CoFixpoint filterp (p:Z -> bool) (s:stream) :=
  let (a,tl) := s in
  if p a then a::filterp p tl
  else
    let (a',tl') := filter1 p tl (Zabs_nat a) in
    a'::filterp p tl'.
```

- Just use a numeric counter, big enough?

The Sieve function

```
CoFixpoint sieve (s : stream) : stream :=
  let (x, s') := s in
  x::sieve (filterp (fun y => Zgt_bool (y mod x) 0) s').

Eval vm_compute in take 10 (sieve (ns 2)).
     = 2::3::5::7::11::13::17::19::23::29::nil
     : list Z
```

- The counter is big enough by "Bertrand, Chebyshev"
- No need for proofs in the definition
- Proving that the stream contains only prime numbers requires inclusion of the Chebyshev result (done by L. Théry in 2003)

Other banned corecursion

```
fib = 0 :: 1 :: (zipWith nat nat plus (tl fib) fib)

fib = 0 :: zipWith nat nat plus fib (1::fib)
```

- "Productive equations"
- Unguarded for syntactic reasons: `tl`, `zipWith`
- Computation of value at rank $n + 2$ only needs values at rank $n + 1$ and n
- Suggests taking an inductive approach, again

Transforming Corecursive functions

- if f represents s, then
  - $\lambda x.\texttt{if } x = 0 \texttt{ then } a \texttt{ else } f\ (x-1)$ represents $a :: s$
  - $f\ 0$ represents $hd\ s$,
  - $\lambda x.\ f\ (x+1)$ represents $tl s$
- On functions from `nat`, `zipWith` $op\ f\ g$ is simply representable by $\lambda x.\ op\ (f\ x)\ (g\ x)$
- Cleanup is required: replace comparison to 0 with match, successor-predecessor, etc...
- Checking guardedness on recursive functions is stronger

Example

```
Fixpoint fibf (n : nat) : Z ;=
  if beq_nat n 0 then 0
  else if beq_nat (n-1) 0 then 1
  else fibf ((n-2) + 1) + fibf (n - 2).
```

Example

```
Fixpoint fibf (n : nat) : Z :=
  match n with
  |      0 => 0
  |    S p => if beq_nat p 0 then 1
             else fibf ((p-1) + 1) + fibf (p - 1)
  end.
```

Example

```
Fixpoint fibf (n : nat) : Z :=
  match n with
  |      0 => 0
  |    S 0 => 1
  |S (S p) => fibf (S p) + fibf p
  end.
```

Example

```
Fixpoint fibf (n : nat) : Z :=
  match n with
  |       0 => 0
  |     S 0 => 1
  | S (S p as q) => fibf q + fibf p
  end.

CoFixpoint repr (f:nat->Z) : stream :=
  f 0%nat :: repr (fun n => f (n+1)%nat).

Definition fib := repr fibf.
```

Mixed inductive and co-inductive types

- Proposed by Altenkirch and Danielsson,
  - Promised as a feature of Agda
- Mark fields of constructors as potentially infinite
- Similar to lazy types of Ocaml
- Programming includes "delay" and "force" primitives
  - potentially infinite data is encapsulated in a delay construct
  - Values can only be retrieved after forcing the computation

Mixed induction and co-induction for the fib problem

- Design ad-hoc co-inductive types, with special constructors to represent function calls
- Trees with connected functions must be well-founded
- Infinite trees with *Well-founded patches*
- Write an interpreter to consume the well-founded patches
  - Terminating recursion requiring induction
- Easy to describe in Altenkirch&Danielsson's language
- Use inductive predicates in Coq
  - Absence of infinite trees of "function calls"

Example of fibonacci and zipWith

```
CoInductive zstream : Type :=
  cstr (x : Z) (s : zstream) | zipPlus (s1 s2 : zstream).

Inductive zf : zstream -> Prop :=
  cz1 : forall x s, zf (cstr x s)
| cz3 : forall s1 s2, zf s1 -> zf s2 ->
        zf (zipPlus s1 s2).
```

- zf expresses the absence of infinite branches from the root
- Capability to produce one value

Correct trees

```
CoInductive zr : zstream -> Prop :=
  cs1 : forall x s, zr s -> zr (cstr x s)
| cs2 : forall s1 s2, zr s1 -> zr s2 -> zf s1 -> zf s2 ->
         zr (zipPlus s1 s2).
```

- `zr` expresses `zf` is always satisfied
- Guarantees productivity forever

Computing a value and the remaining stream

```
Definition ex_zip1 : forall s, zf s -> Z * zstream.
...
```

- ad-hoc recursion on the proof of `zf s`
  - *cf.* Coq'Art, chap. 15, sect. 4
- Easier to define as a proof
- Inversion lemmas require special care

```
Definition qex_zip1 s (h : zr s) : Z * {s' | zr s'}.
```

Removal of all zips

```
CoFixpoint ztostream (s:zstream) (h:zr s) : Stream Z :=
  let (x, (s', hs')) := qex_zip1 s h in
  x::ztostream s' hs'.
```

- Lazy computation, but no true re-use

Instanciation on fib

```
CoFixpoint fib : zstream :=
 cstr 0 (zipPlus fib (cstr 1 fib)).
```

- Easy to prove `zf fib`
- Coinductive proof for `zr fib`, piece of cake
- Then apply `ztostream` to obtain a regular stream
- If all proofs are made transparent, this can be computed

Conclusion

- All parts of this talk have mixed induction and co-induction
- mixed induction and co-induction *à la* Altenkirch&Danielsson should be added to Coq
- The definition relying on `tl` should also be amenable
- Models based on co-inductive data-type plus inductive predicate provide a justification
- But mixed induction and co-induction still lack efficiency