Streams in Coq

Herman Geuvers

Thanks to: Eelis van der Weegen

March 15, 2010

Coinductive data types

Coinductive definitions look very much like inductive definitions:

```
CoInductive stream (T: Set): Set :=
  Cons: T -> stream T -> stream T.
```

`Cons` is the *constructor*.

Coinductive data types

Coinductive definitions look very much like inductive definitions:

```
CoInductive stream (T: Set): Set :=
  Cons: T -> stream T -> stream T.
```

`Cons` is the *constructor*.
But the rules for what is a well-formed object of a coinductive type are different

- Terms of a coinductive type can represent infinite objects.
- These terms are evaluated *lazily*.
- Well-definedness is guaranteed via *guardedness*.

Definition of streams

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

How to define

```
ones : Stream nat
ones = 1 :: ones
```

```
CoFixpoint ones : Stream nat :=
  Cons 1 ones.
```

Definition of streams

```
CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.
```

How to define

```
    ones : Stream nat
    ones = 1 :: ones
```

```
CoFixpoint ones : Stream nat :=
  Cons 1 ones.
```

The recursive call to `ones` is guarded by the constructor `Cons`.
The term `ones` does not reduce to `Cons 1 ones`.

Productivity

```
CoFixpoint ones: Stream nat :=
  Cons 1 ones.
```

The definition of `ones` is *productive*: it is equal to an expression of the form

$$a_0 :: a_2 \ldots :: a_n :: \texttt{ones}$$

with $n > 0$.

Productivity

```
CoFixpoint ones: Stream nat :=
  Cons 1 ones.
```

The definition of `ones` is *productive*: it is equal to an expression of the form

$$a_0 :: a_2 \ldots :: a_n :: \texttt{ones}$$

with $n > 0$.
In Coq, productivity is guaranteed via the syntactic criterion of *guardedness*.

Productivity

```
CoFixpoint ones: Stream nat :=
  Cons 1 ones.
```

The definition of `ones` is *productive*: it is equal to an expression of the form

$$a_0 :: a_2 \ldots :: a_n :: \texttt{ones}$$

with $n > 0$.
In Coq, productivity is guaranteed via the syntactic criterion of *guardedness*.

```
CoFixpoint simple : Stream nat := simple.
```

is not accepted by Coq.

Guardedness

Productivity is not decidable, but guardedness is.
Coq only allows corecursive definitions that are guarded.

Guardedness

Productivity is not decidable, but guardedness is.
Coq only allows corecursive definitions that are guarded.
For streams this means that the *CoFixpoint definition* should look like this:

$$\text{CoFixpoint } s\ \overrightarrow{p} = a_0 :: a_1 :: \ldots : a_n : s\,\overrightarrow{q} \quad \text{with} \quad n > 1$$

Computation

A CoFixpoint definition CoFixpoint $s\ \overrightarrow{p} = \mathsf{Cons}\, a\, (s\, \overrightarrow{q}\,)$ does *not* reduce:

$$s\ \overrightarrow{p} \not\longrightarrow \mathsf{Cons}\, a\, (s\, \overrightarrow{q}\,)$$

Computation

A CoFixpoint definition CoFixpoint $s\,\vec{p} = \mathsf{Cons}\,a\,(s\,\vec{q})$ does *not* reduce:

$$s\,\vec{p} \not\longrightarrow \mathsf{Cons}\,a\,(s\,\vec{q})$$

But it does “under a match”:

$$\mathsf{match}\,s\,\vec{p}\,\mathsf{with}\,\mathsf{Cons}\,x\,t \Rightarrow E(x,t)\,\mathsf{end} \longrightarrow E(a, s\,\vec{q})$$

Filtering

The following definition is not accepted by a Coq

```
CoFixpoint filter (p:A->bool)(s:Stream A) : Stream .
  match s with
     Cons a t => if (p a)
                 then Cons a (filter p t)
                 else (filter p t)
  end.
```

The second occurrence of `filter` is not guarded by a `Cons`.

Destructors on streams

```
Definition hd (s:Stream) := match s with
                            | Cons a t => a
                            end.

Definition tl (s:Stream) := match s with
                            | Cons a t => t
                            end.
```

Destructors on streams

```
Definition hd (s:Stream) := match s with
                              | Cons a t => a
                              end.

Definition tl (s:Stream) := match s with
                              | Cons a t => t
                              end.

Definition head (s:Stream) := let (a,t) := s in a.

Definition tail (s:Stream) := let (a,t) := s in t.
```

Relating streams and functions

There is an isomorphism between `Stream A` and `nat -> A`.

```
CoFixpoint F2S (f:nat->A) : Stream A :=
  Cons (f 0)(F2S (fun n:nat => f (S n))).
```

This just defines

$$F(f) := f(0) :: F(\lambda n.f(n+1))$$

which is correct, because F is guarded by the constructor.

Relating streams and functions

We compute the *n*th tail and the *n*th element of a stream.

```
Fixpoint Str_nth_tl (n:nat) (s:Stream A) : Stream A
  match n with
  | O => s
  | S m => Str_nth_tl m (tl s)
  end.

Definition Str_nth (n:nat) (s:Stream A) : A :=
            hd (Str_nth_tl n s).


Definition S2F (s :Stream A)(n:nat): A :=
            Str_nth n s.
```

This just defines

$$\text{S2F}(s) := \lambda n.\text{hd}(\text{tl}^n(s))$$

Relating streams and functions

We can prove by induction on n

$$\forall n\, \mathrm{S2F}(\mathrm{F2S}(f))(n) = f(n)$$

This is the extensional equality on functions.

Relating streams and functions

We can prove by induction on n

$$\forall n \, \mathrm{S2F}(\mathrm{F2S}(f))(n) = f(n)$$

This is the extensional equality on functions.
How to prove

$$\mathrm{F2S}(\mathrm{S2F}(s)) = s??$$

Relating streams and functions

We can prove by induction on n

$$\forall n \, \mathrm{S2F}(\mathrm{F2S}(f))(n) = f(n)$$

This is the extensional equality on functions.
How to prove

$$\mathrm{F2S}(\mathrm{S2F}(s)) = s??$$

What is the equality on streams?

Digression: Equality on streams

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
    EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

This is the *coinductive equality* on streams.

Digression: Equality on streams

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
    EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

This is the *coinductive equality* on streams.
A proof of `EqSt s1 s2` must have the shape

```
CoFixpoint eq_proof (...) := eqst p eq_proof (...)
```

so the definition of `eq_proof` must be *guarded* by `eqst`.

Examples of Equality on streams

Equality is reflexive:

```
CoFixpoint EqSt_reflex (s : Stream) : EqSt s s :=
 eqst s s (refl_equal (hd s)) (EqSt_reflex (tl s))

  : forall s : Stream, EqSt s s
```

Examples of Equality on streams

Equality is reflexive:

```
CoFixpoint EqSt_reflex (s : Stream) : EqSt s s :=
 eqst s s (refl_equal (hd s)) (EqSt_reflex (tl s))

  : forall s : Stream, EqSt s s

CoFixpoint Eq_sym (s1 s2 : Stream) (H : EqSt s1 s2)
            : EqSt s2 s1 :=
 eqst s2 s1 (let (H0,_) := H in sym_eq H0)
             (Eq_sym (let (_,H1) := H in H1)).

  : forall s1 s2 : Stream, EqSt s1 s2 -> EqSt s2 s1
```

Relating streams and functions

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
    EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

We prove F2S(S2F(s)) $= s$ by

```
CoFixpoint id_on_str (s : Stream A) :
  EqSt (F2S (S2F s)) s :=
  eqst (F2S (S2F s)) s (refl_equal (hd s))
    match s with
    | Cons _ s1 => id_on_str s1
    end.
```

`id_on_str` is guarded by `eqst`.

Relating streams and functions

```
CoInductive EqSt (s1 s2: Stream) : Prop :=
eqst :
 hd s1 = hd s2 ->
    EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

We prove $F(G(s)) = s$ by

```
Lemma ident_on_str : forall s: Stream A, EqSt (F2S
Proof with auto.
cofix ident_on_str.
destruct s.
simpl.
apply identity_on_str.
Qed.
```

Final Coalgebras in Coq

$$
\begin{tikzcd}
X \arrow[r, "\exists f"] \arrow[d, "\langle c, g\rangle"'] & A^{\omega} \arrow[d, "\langle \mathrm{hd}, \mathrm{tl}\rangle"] \\
A \times X \arrow[r, "1 \times f"'] & A \times A^{\omega}
\end{tikzcd}
$$

Final Coalgebras in Coq

$$\begin{tikzcd}
X \arrow[r, "\exists f"] \arrow[d, "\langle c, g\rangle"'] & A^\omega \arrow[d, "\langle \mathrm{hd}, \mathrm{tl}\rangle"] \\
A \times X \arrow[r, "1 \times f"'] & A \times A^\omega
\end{tikzcd}$$

For $c : X \to A$, $g : X \to X$, there is an $f : X \to \text{Stream}A$ such that

- $\forall x : X\,(\mathrm{hd}(f(x)) = c(x))$
- $\forall x : X\,(\mathrm{tl}(f(x)) = f(g(x)))$

Final Coalgebras in Coq

$$\begin{tikzcd}
X \arrow[r, "\exists f"] \arrow[d, "{\langle c, g\rangle}"'] & A^\omega \arrow[d, "{\langle \mathrm{hd}, \mathrm{tl}\rangle}"] \\
A \times X \arrow[r, "1 \times f"'] & A \times A^\omega
\end{tikzcd}$$

For $c : X \to A$, $g : X \to X$, there is an $f : X \to \mathrm{Stream}A$ such that
- $\forall x : X\,(\mathrm{hd}(f(x)) = c(x))$
- $\forall x : X\,(\mathrm{tl}(f(x)) = f(g(x)))$

The first $=$ is on A, the second $=$ is on Stream$A$ (so: `EqSt`).
We call f the *coiteration of c and g*.

Final Coalgebras in Coq

```
CoFixpoint coit : X -> Stream A :=
  fun x: X => Cons (c x) (coit (g x)).

Lemma hd_coit: forall x:X, hd (coit x) = c x.

Lemma tl_coit: forall x:X,
   EqSt (tl (coit x)) (coit (g x)).
```

Final Coalgebras in Coq

$$
\begin{tikzcd}
X \arrow[r, "\exists f"] \arrow[d, "\langle c, g \rangle"'] & A^\omega \arrow[d, "\langle \mathrm{hd}, \mathrm{tl} \rangle"] \\
A \times X \arrow[r, "1 \times f"'] & A \times A^\omega
\end{tikzcd}
$$

Final Coalgebras in Coq

$$\begin{tikzcd}
X \arrow[r, "\exists f"] \arrow[d, "\langle c, g\rangle"'] & A^\omega \arrow[d, "\langle \mathrm{hd}, \mathrm{tl}\rangle"] \\
A \times X \arrow[r, "1 \times f"'] & A \times A^\omega
\end{tikzcd}$$

In final coalgebras: $f := \mathrm{coit}\, c\, g$ is *unique*. Can we prove that in Coq? If $\forall x : X\, (\mathrm{hd}(f'(x)) = c(x))$ and $\forall x : X\, (\mathrm{tl}(f'(x)) = f'(g(x)))$, then $\forall x : X\, (f'(x) = \mathrm{coit}\, c\, g(x))$.

$$\begin{tikzcd} X \arrow[r, "\exists f"] \arrow[d, "\langle c, g\rangle"'] & A^\omega \arrow[d, "\langle \mathrm{hd}, \mathrm{tl}\rangle"] \\ A \times X \arrow[r, "1 \times f"'] & A \times A^\omega \end{tikzcd}$$

In final coalgebras: $f := \mathrm{coit}\, c\, g$ is *unique*. Can we prove that in Coq? If $\forall x : X\, (\mathrm{hd}(f'(x)) = c(x))$ and $\forall x : X\, (\mathrm{tl}(f'(x)) = f'(g(x)))$, then $\forall x : X\, (f'(x) = \mathrm{coit}\, c\, g(x))$.

```
Lemma unique: forall f : X -> Stream A,
  (forall x:X, hd (f x) = c x) ->
  (forall x:X, EqSt (tl (f x)) (f (g x))) ->
  (forall x:X, EqSt (f x)(coit x)).
```

We can prove this

Bisimulation and Stream equality in Coq

If two streams are bisimilar, they are stream-equal.

```
Variable R: (Stream A) -> (Stream A) -> Prop.
Hypothesis bisim: forall s t, R s t ->
    hd s = hd t /\ R(tl s)(tl t).

Lemma bisim_implies_EqSt : forall s t, R s t ->
    EqSt s t.
```

Bisimulation and Stream equality in Coq

If two streams are bisimilar, they are stream-equal.

```
Variable R: (Stream A) -> (Stream A) -> Prop.
Hypothesis bisim: forall s t, R s t ->
    hd s = hd t /\ R(tl s)(tl t).

Lemma bisim_implies_EqSt : forall s t, R s t ->
    EqSt s t.
```

If two streams are stream-equal, they are bisimilar.

```
Lemma EqSt_implies_bisim :
 exists Q : Stream A -> Stream A ->Prop,
 (forall s t, Q s t  ->  hd s = hd t /\ Q(tl s)(tl
     /\  forall s t, EqSt s t ->  R s t.
```

End of Part one

This is the end of part one.

The Hamming Stream in Coq (Eelis vd Weegen)

- All natural numbers of the form $2^i 3^j 5^k$
- $\ldots$ in increasing order.

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, \ldots$$

The Hamming Stream in Coq (Eelis vd Weegen)

- All natural numbers of the form $2^i 3^j 5^k$
- ... in increasing order.

$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, \ldots$

Popularized by Edsger Dijkstra

Formalisation

We would like to

- Write a formal specification for the Hamming stream
- Give an implementation of the Hamming stream (an algorithm, e.g. Dijkstra's)
- Prove that the implementation satisfies the specification
- All this formally in Coq ...

Dijkstra's Hamming-Stream Algorithm

Algorithm uses the `merge` function

```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x:xs) (y:ys)
  | x < y    = x : merge xs (y:ys)
  | x > y    = y : merge (x:xs) ys
  | x == y   = x : merge xs ys
```

Dijkstra's Hamming-Stream Algorithm

Algorithm uses the `merge` function

```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x:xs) (y:ys)
  | x < y    = x : merge xs (y:ys)
  | x > y    = y : merge (x:xs) ys
  | x == y   = x : merge xs ys


hamming :: [Integer]
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

Dijkstra's Hamming-Stream Algorithm

```
hamming :: [Integer]
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

- Functional
- Efficient
- Elegant(?)
- Representation of streams via the usual list datatype

Formal specification

A stream *s* is the Hamming stream if

- when a number occurs in *s*, it has the right shape ("soundness");
- when a number has the right shape, it occurs in *s* ("completeness");
- *s* is increasing.

```
Record hamming_stream: Set :=
{ s :      Stream nat
; s_sound: everywhere smooth s
; s_comp : forall n, smooth n -> in_stream (eq n) s
; s_incr : increases s
}.
```

Dijkstra's Hamming-stream algorithm

It is not possible to use Dijkstra's Hamming-stream algorithm directly in Coq.
Problem: Coq requires a CoFixpoint definition to be *explicitly guarded*. Dijkstra's Hamming-stream algoritme niet guarded:

Dijkstra's Hamming-stream algorithm

It is not possible to use Dijkstra's Hamming-stream algorithm directly in Coq.
Problem: Coq requires a CoFixpoint definition to be *explicitly guarded*. Dijkstra's Hamming-stream algoritme niet guarded:

```
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

Dijkstra’s Hamming-stream algorithm

It is not possible to use Dijkstra’s Hamming-stream algorithm directly in Coq.
Problem: Coq requires a CoFixpoint definition to be *explicitly guarded*. Dijkstra’s Hamming-stream algoritme niet guarded:

```
hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming)
      (map (* 5) hamming) )
```

"map" and "merge" could do anything, e.g. take the tail of a list ...

```
CoFixpoint ff (s:Stream nat) :=
  Cons 1 (tl (ff s)).
```

is ill-formed.

Another Hamming-stream algorithm

How to formalize the Hamming stream?

- Use another algorithm that is (syntactically) guarded.
- Tricks to use Dijkstra's algorithm after all.

Another Hamming-stream algorithm

```
CoFixpoint ham_from (l: ne_list nat): Stream nat :=
  Cons (head l) (ham_from (
   enqueue (head l * 2)(
     enqueue (head l * 3)(
       enqueue (head l * 5)(tail l)))))
```

where `enqueue` takes an *n* and an increasing list *l* and puts *n* "at the right place" in *l*:

```
Fixpoint enqueue(n: nat)(l: list nat) : ne_list nat
  match l with
  | nil => [n]
  | h :: t =>
    if n<h then n :: l else
     if n=h then h :: t else
       h :: (enqueue n t)
    end.
```

The orginal Hamming-stream algorithm?

Define hamming and merge on lists:

```
Definition ham_body (l: list nat): list nat :=
cons 1 (merge (map (mult 2) l)
 (merge (map (mult 3) l) (map (mult 5) l))).
```

The orginal Hamming-stream algorithm?

Define hamming and merge on lists:

```
Definition ham_body (l: list nat): list nat :=
cons 1 (merge (map (mult 2) l)
 (merge (map (mult 3) l) (map (mult 5) l))).
```

```
n | repeat_apply ham_body nil n
-----------------------------
0 | 1
1 | 1, 2, 3, 5
2 | 1, 2, 3, 4, 5, 6, 9, 10, 15, 25
3 | 1, 2, 3, 4, 5, 6, 8, 9,  10, 12, 15, 18, 20, 25
```

Taking the "diagonal"produces the Hamming stream.

Some conclusions

- Coq can serve as a general framework for specifying and implementing streams and for proving that implementations satisfy their specifications.
- The implementation language is restricted (only guarded definitions, only terminating algorithms), so one cannot "just" write the Haskell algorithm in Coq.
- Ad hoc and more generic approaches exist to encode Hamming, Fib, Thue-Morse and to prove their correctness.