Coinductive definitions look very much like inductive definitions:

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

Cons is the constructor.
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.
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.
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.
CoFixpoint ones : Stream nat := 
Cons 1 ones.

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

\[ a_0 :: a_2 \ldots :: a_n :: \text{ones} \]

with \( n > 0 \).
CoFixpoint ones: Stream nat :=
    Cons 1 ones.

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

\[ a_0 :: a_2 \ldots :: a_n :: \text{ones} \]

with \( n > 0 \).

In Coq, productivity is guaranteed via the syntactic criterion of guardedness.
CoFixpoint ones : Stream nat :=
  Cons 1 ones.

The definition of ones is \textit{productive}: it is equal to an expression of the form

\[ a_0 :: a_2 \ldots :: a_n :: \text{ones} \]

with \( n > 0 \).

In Coq, productivity is guaranteed via the syntactic criterion of \textit{guardedness}.

CoFixpoint simple : Stream nat := simple.

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

\[
\text{CoFixpoint } s \overset{\rho}{\rightarrow} = a_0 :: a_1 :: \ldots :: a_n : s \overset{\varrho}{\rightarrow} \quad \text{with} \quad n > 1
\]
A CoFixpoint definition \( \text{CoFixpoint } s \xrightarrow{p} = \text{Cons } a (s \xrightarrow{q}) \) does not reduce:

\[
s \xrightarrow{p} \not\xrightarrow{} \text{Cons } a (s \xrightarrow{q})
\]
A CoFixpoint definition $\text{CoFixpoint } s \rightarrow p = \text{Cons } a (s \rightarrow q)$ does not reduce:

$$s \rightarrow p \not\rightarrow \text{Cons } a (s \rightarrow q)$$

But it does “under a match”:

match $s \rightarrow p$ with Cons $x$ $t$ $\Rightarrow$ $E(x, t)$ end $\rightarrow$ $E(a, s \rightarrow q)$
The following definition is not accepted by a Coq

CoFixpoint filter (p:A->bool)(s:Stream A) : Stream A :=
  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.
There is an isomorphism between $\text{Stream } A$ and $\text{nat} \rightarrow A$.

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.

\[
\text{Fixpoint \( \text{Str\_nth\_tl} \) \( n: \text{nat} \) \( s: \text{Stream} \ A \) : Stream A :=
\begin{align*}
\text{match} & \ n \ \text{with} \\
\text{\|} \ O & \Rightarrow s \\
\text{\|} \ S \ m & \Rightarrow \text{Str\_nth\_tl} \ m \ (\text{tl} \ s) \\
\text{end.}
\end{align*}
\]

\[
\text{Definition \( \text{Str\_nth} \) \( n: \text{nat} \) \( s: \text{Stream} \ A \) : A :=}
\begin{align*}
\text{hd} & \ (\text{Str\_nth\_tl} \ n \ s).
\end{align*}
\]

\[
\text{Definition \( \text{S2F} \) \( s : \text{Stream} \ A \)(n: \text{nat}) : A :=}
\begin{align*}
\text{Str\_nth} \ n \ s
\end{align*}
\]

This just defines

\[
\text{S2F}(s) := \lambda n.\text{hd}(\text{tl}^n(s))
\]
We can prove by induction on $n$

$$\forall n S2F(F2S(f))(n) = f(n)$$

This is the extensional equality on functions.
We can prove by induction on $n$

$$\forall n \ S2F(F2S(f))(n) = f(n)$$

This is the extensional equality on functions.
How to prove

$$F2S(S2F(s)) = s$$
We can prove by induction on $n$

$$\forall n \ S2F(F2S(f))(n) = f(n)$$

This is the extensional equality on functions.
How to prove

$$F2S(S2F(s)) = s$$

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

This is the \textit{coinductive equality} on streams. A proof of \textit{EqSt s1 s2} must have the shape

\text{CoFixpoint eq\_proof (\ldots) := eqst p eq\_proof (\ldots)}

so the definition of \textit{eq\_proof} must be \textit{guarded} by \textit{eqst}. 
Examples of Equality on streams

Equality is reflexive:

\[
\text{CoFixpoint EqSt_reflex (s : Stream) : EqSt s s := eqst s s (refl_equal (hd s)) (EqSt_reflex (tl s))}
\]

: forall s : Stream, EqSt s s

Herman Geuvers
Streams in Coq
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
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.
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 (S2F s))
Proof with auto.
cofix ident_on_str.
destruct s.
simpl.
apply identity_on_str.
Qed.
For \(c : X \rightarrow A\), \(g : X \rightarrow X\), there is an \(f : X \rightarrow \text{Stream } A\) such that:

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

We call \(f\) the coiteration of \(c\) and \(g\).
For $c : X \rightarrow A$, $g : X \rightarrow X$, there is an $f : X \rightarrow \text{Stream}A$ such that

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

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

The first \( = \) is on \( A \), the second \( = \) is on \( \text{Stream}A \) (so: \( \text{EqSt} \)).

We call \( f \) the \textit{coiteration of} \( c \) and \( g \).
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

In final coalgebras:

\[ f := \text{coit } c \, g \]
is unique. Can we prove that in Coq? If

\[ \forall x : X, \text{hd} (f' (x)) = c (x) \]
and

\[ \forall x : X, \text{tl} (f' (x)) = f' (g (x)) \],

then

\[ \forall x : X, f' (x) = \text{coit } c \, g (x) \].

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

Lemma unique: \( \forall f : X \rightarrow \text{Stream} \ A, \)

\[
(\forall x : X, \ \text{hd} \ (f \ x) = c \ x) \rightarrow
(\forall x : X, \ \text{EqSt} \ (\text{tl} \ (f \ x)) \ (f \ (g \ x))) \rightarrow
(\forall x : X, \ \text{EqSt} \ (f \ x)(\text{coit} \ x)).
\]

We can prove this.
If two streams are bisimilar, they are stream-equal.

Variable \( R: (\text{Stream A}) \rightarrow (\text{Stream A}) \rightarrow \text{Prop} \).
Hypothesis \( \text{bisim}: \forall s \ t, R \ s \ t \rightarrow \)
\( \text{hd } s = \text{hd } t \land R(\text{tl } s)(\text{tl } t) \).

Lemma \( \text{bisim_implies_EqSt} : \forall s \ t, R \ s \ t \rightarrow \)
\( \text{EqSt } s \ t \).
If two streams are bisimilar, they are stream-equal.

Variable $R : (\text{Stream } A) \rightarrow (\text{Stream } A) \rightarrow \text{Prop}$. Hypothesis \textit{bisim}:\ $\forall s \ t, R\ s\ t \rightarrow$
\hspace{1cm} $\text{hd } s = \text{hd } t \ \land \ R(\text{tl } s)(\text{tl } t)$.

Lemma \textit{bisim_implies_EqSt} : $\forall s \ t, R\ s\ t \rightarrow\ EqSt\ s\ t$.

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

Lemma \textit{EqSt_implies_bisim} :
\hspace{1cm} exists $Q : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Prop}$,
\hspace{1cm} $(\forall s \ t, Q\ s\ t \rightarrow\ \text{hd } s = \text{hd } t \ \land \ Q(\text{tl } s)(\text{tl } t)$
\hspace{2cm} $\land\ \forall s \ t, EqSt\ s\ t \rightarrow R\ s\ t$. 
This is the end of part one.
All natural numbers of the form $2^i3^j5^k$

... in increasing order.

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, ...
All natural numbers of the form $2^i3^j5^k$

... in increasing order.

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

Popularized by Edsger Dijkstra
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

```haskell
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
```

Herman Geuvers
Streams in Coq
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

```haskell
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
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
}.
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 algorithm is not guarded:

\[
\text{hamming} = 1 : \text{merge} (\text{map} \ ((\ast \times 2)) \ \text{hamming}) \ \text{merge} (\text{map} \ ((\ast \times 3)) \ \text{hamming}) \ \text{merge} (\text{map} \ ((\ast \times 5)) \ \text{hamming})
\]
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:

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

Herman Geuvers

Streams in Coq
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:

\[
\text{hamming} = 1 : \\
\text{merge} (\text{map} \ (* \ 2 \ \text{hamming}) \ (\\
\text{merge} (\text{map} \ (* \ 3 \ \text{hamming}) \\
(\text{map} \ (* \ 5 \ \text{hamming}) ) \\
\text{"map" and "merge" could do anything, e.g. take the tail of a list...} \\
\text{CoFixpoint ff (s:Stream nat) := Cons 1 (tl (ff s)).} \\
\text{is ill-formed.}
\]
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.
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))).
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.
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.