

Defining Streams in Agda using Copatterns and Sized Types

Andreas Abel

Workshop *Representing Streams II*, January 2014

```
{-# OPTIONS -copatterns -sized-types #-}

open import Size
open import Function
open import Relation.Binary.PropositionalEquality
open import Relation.Nullary using (Dec; yes; no)
open import Relation.Nullary.Decidable using ([_])

open import Data.Bool using (Bool; true; false; if _ then _ else _)
open import Data.List using (List; module List; []; _::_; _++_)
open import Data.Nat using (ℕ; zero; suc; _*_ ; _≤? _; compare; less; equal; greater)
open import Data.Product using (_×_ ; _,_ ; _,'_ ; proj1; proj2)

- Sized streams via head/tail.

record Stream {i : Size} (A : Set) : Set where
  coinductive
  constructor _::_
  field
    head : A
    tail : ∀ {j : Size < i} → Stream {j} A
open Stream public

- Functoriality.

map : ∀ {i A B} (f : A → B) (s : Stream {i} A) → Stream {i} B
head (map f s) = f (head s)
tail (map {i} f s) {j} = map {j} f (tail s {j})

zipWith : ∀ {i A B C} (f : A → B → C) → Stream {i} A → Stream {i} B → Stream {i} C
head (zipWith f s t) = f (head s) (head t)
tail (zipWith f s t) = zipWith f (tail s) (tail t)
```

- Generating a stream by replication.

```
repeat :  $\forall \{i\} A (a : A) \rightarrow \text{Stream } \{i\} A$   
head (repeat a) = a  
tail (repeat a) = repeat a
```

- Generating a stream from a coalgebra.

```
unfold :  $\forall \{i\} \{A S : \text{Set}\} (step : S \rightarrow A \times S) (s : S) \rightarrow \text{Stream } \{i\} A$   
head (unfold step s) = proj1 (step s)  
tail (unfold step s) = unfold step (proj2 (step s))
```

- Alternating elements of two streams.

```
interleave :  $\forall \{i\} A \rightarrow \text{Stream } \{i\} A \rightarrow \text{Stream } \{i\} A \rightarrow \text{Stream } \{i\} A$   
head (interleave s t) = head s  
tail (interleave s t) = interleave t (tail s)
```

- A slightly more precise type (but harder for Agda to infer hidden args).

```
interleave' :  $\forall \{i\} A$   
          ( $s : \text{Stream } \{i\} A$ ) ( $t : \{j : \text{Size} < i\} \rightarrow \text{Stream } \{j\} A$ )  $\rightarrow \text{Stream } \{i\} A$   
head (interleave' s t) = head s  
tail (interleave' s t) = interleave' t (tail s)
```

- Substreams

mutual

```
evens :  $\forall \{i\} A \rightarrow \text{Stream } A \rightarrow \text{Stream } \{i\} A$   
head (evens s) = head s  
tail (evens s) = odds (tail s)
```

```
odds :  $\forall \{i\} A \rightarrow \text{Stream } A \rightarrow \text{Stream } \{i\} A$   
odds s = evens (tail s)
```

- Streams and lists.

- Prepending a list to a stream.

```
++s :  $\forall \{i\} A \rightarrow \text{List } A \rightarrow \text{Stream } \{i\} A \rightarrow \text{Stream } \{i\} A$   
[] ++s s = s  
(a :: as) ++s s = a :: (as ++s s)
```

- Taking an initial segment of a stream.

```

takes : ∀ {A} (n : ℕ) (s : Stream A) → List A
takes 0      s = []
takes (suc n) s = head s :: takes n (tail s)

```

- Unfold which produces several outputs at one step

```

record List1 (A : Set) : Set where
  constructor _::__
  field
    first : A
    rest  : List A
open List1 public

```

```

unfold+ : ∀ {ℓ} {S : Size → Set} {A : Set}
  (step : ∀ {i} → S i → List1 A × (∀ {j : Size < i} → S j)) →
  ∀ {i} → (s : S i) → Stream {i} A

```

```

head (unfold+ step s) = first (proj1 (step s))
tail (unfold+ step s) = let (_ :: l, s') = step s
                        in l ++s unfold+ step s'

```

- Ordered merge.

```

merge : ∀ {i} (s t : Stream {i} ℕ) → Stream {i} ℕ
head (merge s t) = if [ head s ≤? head t ] then head s else head t
tail (merge s t) = if [ head s ≤? head t ] then
                    if [ head t ≤? head s ]
                      then merge (tail s) (tail t) - eliminate duplicates!
                      else merge (tail s) t
                    else merge s (tail t)

```

- Hamming stream.

```

hamming : ∀ {i} → Stream {i} ℕ
head (hamming) = 1
tail (hamming {i}) {j} =
  (merge {j} (map {j} (λ x → 2 * x) (hamming {j})))
  (merge {j} (map {j} (λ x → 3 * x) (hamming {j})))
  (map {j} (λ x → 5 * x) (hamming {j})))

```