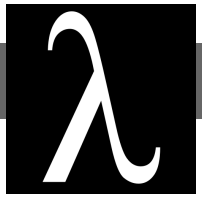

Contents

1 Preamble

Streams are potentially infinite lists whose elements are computed lazily. This means that the elements of a stream are determined by **suspended computations** that generate the next element only when forced to do so. We achieve lazy evaluation using "thunks," which are function values whose bodies only evaluate when applied to the argument `unit`.

The type `'a Stream.stream` represents a suspension of the type `'a Stream.front`, through which the elements of the stream can be exposed.

Unless noted otherwise, all functions in this library are **maximally lazy**, meaning their values are not computed until absolutely necessary.

2 Signature

```
signature STREAM =
sig
  type 'a stream   (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  exception EmptyStream


  (* Lazy Stream Delay and Exposure *)

  val delay : (unit -> 'a front) -> 'a stream
  val expose : 'a stream -> 'a front


  (* Stream Construction *)

  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val fromList : 'a list -> 'a stream
  val tabulate : (int -> 'a) -> 'a stream


  (* Deconstructing a Stream *)

  val null : 'a stream -> bool
  val hd : 'a stream -> 'a
  val take : 'a stream * int -> 'a list
  val toList : 'a stream -> 'a list


  (* Simple Transformations *)

  val tl : 'a stream -> 'a stream
  val drop : 'a stream * int -> 'a stream
  val append : 'a stream * 'a stream -> 'a stream


  (* Combinators and Higher-Order Functions *)

  val map : ('a -> 'b) -> 'a stream -> 'b stream
  val filter : ('a -> bool) -> 'a stream -> 'a stream
  val zip : 'a stream * 'b stream -> ('a * 'b) stream

end
```

3 Documentation

Constraint: Whenever you use these stream functions, please make sure you meet the specified preconditions.

If you do not meet the precondition for a function, it may not behave as expected.

3.1 Lazy Stream Delay and Exposure

```
delay : (unit -> 'a front) -> 'a stream
```

ENSURES: `delay f` evaluates to a stream version of the suspended front.

```
expose : 'a stream -> 'a front
```

ENSURES: `expose s` evaluates to a front version of the stream.

Note that `expose (delay f)` $\Longrightarrow$ `f ()`.

3.2 Stream Construction

```
empty : 'a stream
```

ENSURES: `empty` is a stream with no elements.

```
cons : 'a * 'a stream -> 'a stream
```

ENSURES: `cons (x,s)` evaluates to a stream whose first item is `x` and whose remaining items are exactly the stream `s`.

```
fromList : 'a list -> 'a stream
```

ENSURES: `fromList L` returns a stream consisting of the elements of `L`, preserving order. This function is intended primarily for debugging purposes.

```
tabulate : (int -> 'a) -> 'a stream
```

ENSURES: `tabulate f` evaluates to a stream `s` where the $\texttt{i}^{\text{th}}$ element of `s` is equal to `f i`.

Note that indices are zero-indexed.

3.3 Deconstructing a Stream

```
null : 'a stream -> bool
```

ENSURES: `null s` evaluates to `true` if `s` is an empty stream, and `false` otherwise.

```
hd : 'a stream -> 'a
```

ENSURES: `hd s` evaluates to the first element of `s` if `s` is non-empty. `hd s` raises `EmptyStream` otherwise.

```
take : 'a stream * int -> 'a list
```

REQUIRES: `i` ≥ 0

ENSURES: `take (s, i)` evaluates to the list containing exactly the first `i` elements of `s` if `s` contains at least `i` elements, and raises `Subscript` otherwise.

```
toList : 'a stream -> 'a list
```

ENSURES: `toList s` returns a list consisting of the elements of `s`, preserving order. This function will loop forever if `s` is an infinite stream. This function is intended primarily for debugging purposes.

3.4 Simple Transformations

```
tl : 'a stream -> 'a stream
```

ENSURES: `tl s` evaluates to the stream containing all but the first element of `s` if `s` is non-empty. `tl s` raises `EmptyStream` otherwise.

```
drop : 'a stream * int -> 'a stream
```

REQUIRES: `i` ≥ 0

ENSURES: `drop (s, i)` evaluates to the stream containing all but the first `i` elements of `s` if `s` contains at least `i` elements, and raises `Subscript` otherwise.

```
append : 'a stream * 'a stream -> 'a stream
```

ENSURES: `append (s1,s2)` $\Longrightarrow$ `s`, where `s` is `s2` appended to `s1`.

3.5 Combinators and Higher-Order Functions

```
map : ('a -> 'b) -> 'a stream -> 'b stream
```

ENSURES: `map f s` $\Longrightarrow$ `s'`, where each element `x` in `s` corresponds to `f x` in `s'`.

```
filter : ('a -> bool) -> 'a stream -> 'a stream
```

ENSURES: `filter p s` evaluates to a stream containing all of the elements `x` of `s` such that `p x` $\Longrightarrow$ `true`, preserving element order.

```
zip : 'a stream * 'b stream -> ('a * 'b) stream
```

ENSURES: `zip (s1,s2)` $\Longrightarrow$ `s'`, where the `i`$^{\text{th}}$ element of `s'` is the pair of the `i`$^{\text{th}}$ element of `s1` and the `i`$^{\text{th}}$ element of `s2`.

`s'` contains as many elements as the stream (`s1` or `s2`) with fewer elements.