Lazy Lists in SML

Sieve of Eratosthenes

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling
Technical University of Denmark
Lazy Lists

- *lazy evaluation* or *delayed evaluation* is the technique of delaying a computation until the result of the computation is needed.

  Default in lazy languages like Haskell

A special form of this is *lazy lists*, where the elements are not evaluated until their values are required by the rest of the program.

- *lazy lists* may be infinite
Lazy Lists

- *lazy evaluation* or *delayed evaluation* is the technique of delaying a computation until the result of the computation is needed.

  Default in lazy languages like Haskell

A special form of this is *lazy lists*, where the elements are not evaluated until their values are required by the rest of the program.

- *lazy lists* may be infinite
  - a finite part of a lazy list may be used in computations
Lazy Lists

- *lazy evaluation* or *delayed evaluation* is the technique of delaying a computation until the result of the computation is needed.

  Default in lazy languages like Haskell

A special form of this is *lazy lists*, where the elements are not evaluated until their values are required by the rest of the program.

- *lazy lists* may be infinite
  a finite part of a lazy list may be used in computations

Example:
- Consider the sequence of all prime numbers
- the first 5 are 2,3,5,7,11

Sieve of Eratosthenes
Lazy Lists in SML

A lazy list or *sequence* is represented in SML by the head of the sequence, and a function for computing its (possibly infinite) tail:

```sml
datatype 'a seq = Empty
    | Cons of 'a * (unit -> 'a seq);

The function `seqFrom i` represents the sequence `i, i + 1, i + 2, ...`:

```sml
fun seqFrom i = Cons(i, fn () => seqFrom(i+1));
```

- the delay of the computation of `i + 1, i + 2, ...` is obtained by the function `fn () => seqFrom(i + 1)`
Head and Tail of sequences:

fun hdSeq(Cons(x, _)) = x;

fun tlSeq(Cons(_, xt)) = xt();

Examples:

val nat = seqFrom 0;
> val nat = Cons(0, fn) : int seq

hdSeq nat;
> vat it = 0 : int

tlSeq nat;
> val it = Cons(1, fn) : int seq
Functions on sequences (II)

Take and drop elements of sequences:

```ml
fun takeSeq(0, _) = []
  | takeSeq(_, Empty) = []
  | takeSeq(i, Cons(n,xt)) = n :: takeSeq(i-1, xt());

fun dropSeq(0, xs) = xs
  | dropSeq(i, Cons(_, xt)) = dropSeq(i-1, xt());

takeSeq(5, nat);
> val it = [0,1,2,3,4] : int list

dropSeq(5, nat);
> val it = Cons(5, fn) : int seq
```
Functions on sequences (III)

A higher-order function on sequences:

```haskell
fun filterSeq p Empty       = Empty
  | filterSeq p (Cons(x, xt)) =
     if p x then Cons(x, fn () => filterSeq p (xt()))
     else filterSeq p (xt());

val even = filterSeq (fn n => n mod 2 = 0) nat;

takeSeq(5, even);
> val it = [0, 2, 4, 6, 8] : int list
```
Sieve of Eratosthenes

Greek mathematician (194 – 176 BC)

Computation of prime numbers

- start with the sequence $2, 3, 4, 5, 6, ...$
  select head ($2$), and remove multiples of 2 from the sequence

- next sequence $3, 5, 7, 9, 11, ...$
  select head ($3$), and remove multiples of 3 from the sequence

- next sequence $5, 7, 11, 13, 17, ...$
  select head ($5$), and remove multiples of 5 from the sequence

- ...
Sieve of Eratosthenes in SML

Remove multiples of \(a\) from sequence \(ns\):

\[
\text{fun sift } a \text{ ns } = \text{filterSeq (fn n } => \text{ n mod a } \neq 0) \text{ ns;}
\]

Select head and remove multiples of head from the tail – recursively:

\[
\text{fun sieve(Cons(n, nt)) } = \\
\quad \text{Cons(n, fn () } => \text{ sieve(sift n (nt())));}
\]

The sequence of prime numbers and the \(n\)'th prime number:

\[
\text{val primes } = \text{sieve(seqFrom 2);} \\
\text{fun primeN n } = \text{hdSeq(dropSeq(n-1, primes));}
\]

\[
\text{primeN 1000;} \\
\text{> val it } = 7919 : \text{ int}
\]