Functional Programming

Lecture 7 — Infinite lists and lazy evaluation

Pedro Vasconcelos
DCC/FCUP

2025

Infinite lists

We can use lists to represent finite sequences, e.g.:

```
[1,2,3,4] = 1:2:3:4:[]
```

In this lecture we will see that we can use lists to represent infinite sequences, e.g.:

```
[1..] =  1:2:3:4:5:...
```

We cannot write an infinite lists by extension; we will use enumerations, list comprehensions or recursive definitions.

Examples

```
-- natural numbers e.g. non-negative integers
nats :: [Integer]
nats = [0..]

-- non-negative even numbers
evens :: [Integer]
evens = [0,2..]

-- the infinite list of ones
ones :: [Integer]
ones = 1 : ones

-- infinite list of integers from a number
intsFrom :: Integer → [Integer]
intsFrom n = n : intsFrom (n+1)
```

Computing with infinite lists

Because of lazy evaluation, infinite lists are computed on demand and only as far as necessary.

```
  head ones
=
  head (1:ones)
=
  1
```

Computing with infinite lists (cont.)

A computation that requires traversing the entire infinite list will never terminate.

```
  length ones
=
  length (1:ones)
=
  1 + length ones
=
  1 + length (1:ones)
=
  1 + (1 + length ones)
=
  ⋮
```

does not terminate

Producing infinite lists

Many functions from the Prelude produce infinite lists when supplied with an infinite list as input:

```
ghci> map (2*) [1..]
[2, 4, 6, 8, 10, ...

ghci> filter (\x → x`mod`2/=0) [1..]
[1, 3, 5, 7, 9, ...
```

We could also use list comprehensions:

```
> [2*x | x←[1..]]
[2, 4, 6, 8, 10, ...

> [x | x←[1..], x`mod`2/=0]
[1, 3, 5, 7, 9, ...
```

Producing infinite lists (cont.)

The following functions produce specifically produce infinite lists:

```
repeat :: a → [a]
-- repeat x = x:x:x:...

cycle :: [a] → [a]
-- cycle xs = xs++xs++xs++...

iterate :: (a → a) → a → [a]
-- iterate f x = x : f x : f(f x) : f(f(f x)) : ...
```

Note that `iterate` is a higher-order function because its first argument is a function.

Producing infinite lists (cont.)

Testing in GHCi:

```
> take 10 (repeat 1)
[1,1,1,1,1,1,1,1,1,1]

> take 10 (repeat 'a')
"aaaaaaaaaa"

> take 10 (cycle [1,-1])
[1,-1,1,-1,1,1,-1,1,-1,1]

> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512]
```

Producing infinite lists (cont.)

These functions are defined in the Prelude using recursion:

```
repeat :: a → [a]
repeat x = xs where xs = x:xs

cycle :: [a] → [a]
cycle [] = error "empty list"
cycle xs = xs' where xs' = xs++xs'

iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

Using infinite lists

- Infinite lists are useful for simplifying the processing of finite lists
- They allow separating the generation from the consumption of a sequence
- This allows greater modularity in program decomposition

Example 1

Padding a string

Write a function

```
padding :: Int → String → String
```

that pad a string with spaces so that it takes exactly *w* characters (the first argument).

If the string has fewer than *w* characters, then space should be added to the end.

If the string has more than *w* characteres, then it should be truncated.

Example 1 (cont.)

Padding a string

Examples

```
> padding 10 "Haskell"
"Haskell   "

> padding 10 "Haskell B. Curry"
"Haskell B."
```

Example 1 (cont.)

Padding a string

A complicated solution that tests whether we need to add spaces or truncate the string:

```
padding w xs
    | k < w     = xs ++ replicate (w-k) ' '
    | otherwise = take w xs
    where k = length xs
```

A simpler solution using concatenation with an infinite list:

```
padding w xs = take w (xs++repeat ' ')
```

Example 2

Approximating square roots

Compute an approximation to $\sqrt{q}$:

1. Start with $x_0 = q$
2. At each iteration, improve the approximation taking

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{q}{x_n}\right)$$

3. We can generate an infinite sequence of approximations and define a termination condition separately:

   number of iterations stop after a given number of iterations

   absolute error stop when the distance between approximations is smaller than a given tolerace

Example 2 (cont.)

Approximating square roots

```
-- infinite sequence of square root approximations
approximations :: Double → [Double]
approximations q = iterate (\x → 0.5*(x+q/x)) q

-- stop after a given number of iterations
iterations :: [Double] → Int → Double
iterations xs n = xs !! n

-- stop with an absolute diference criteria
absoluteDiff :: [Double] → Double → Double
absoluteDiff xs eps
    = head [x' | (x,x')←zip xs (tail xs)
               , abs(x-x')<eps]
```

Example 2 (cont.)

Approximating square roots

Examples for approximating $\sqrt{2}$:

```
> approximations 2.0
[2.0,1.5,1.4166667,1.4142157, 1.4142135, 1.4142135,

> approximations 2.0 `iterations` 5
1.4142135

> approximations 2.0 `absoluteDiff` 0.01
1.4166667

> approximations 2.0 `absoluteDiff` 0.001
1.4142135
```

Example 3

The Fibonacci sequence

- Starts with 0 and 1
- Each following value the *sum of the previous two values*

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots, a, b, a{+}b, \ldots$$

Example 3 (cont.)

The Fibonacci sequence

In Haskell we can define the infinite Fibonacci sequence recursively:

```
fibs :: [Integer]
fibs = 0 : 1 : [a+b | (a,b)←zip fibs (tail fibs)]
```

Alternative using the `zipWith` function (see the exercise sheet):

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Example 3 (cont.)

The Fibonacci sequence

The first 10 Fibonacci numbers:

```
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

The 9th Fibonacci number (indices begin at zero):

```
> fibs!!8
21
```

The first Fibonacci number greater than 100:

```
> head (dropWhile (<=100) fibs)
144
```

Example 4

Infinite lists of prime numbers

Generate the infinite list of prime numbers using a "pseudo sieve of Erathostenes":

1. Start with the list $[2, 3, 4, \ldots]$ of all numbers from 2;
2. Mark the first number p in the list as prime;
3. Remove p from the list as well as all its multiples;
4. Repeat step 2.

Note that step 3 envolves processing an infinite list, but can it still terminate!

Example 4 (cont.)
Infinite lists of prime numbers

Solution in Haskell:

```
primes :: [Integer]
primes = sieve [2..]

sieve :: [Integer] → [Integer]
sieve (p:xs) = p : sieve [x | x←xs, x`mod`p/=0]
```

Example 4 (cont.)

Infinite lists of prime numbers

Generate the first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

How many primes less than 100 are there?

```
> length (takeWhile (<100) primes)
25
```