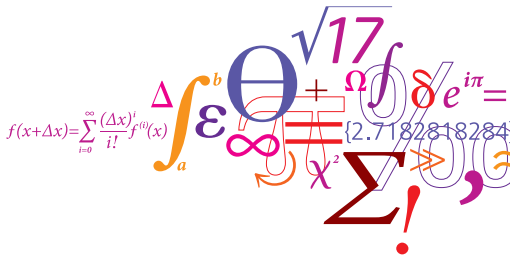


02157 Functional Programming

Sequences

Michael R. Hansen



DTU Informatics

Department of Informatics and Mathematical Modelling

Sequences (or 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

It is occasionally efficient to be lazy.

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

- a *sequence* may be infinite
just a finite part of a it is used in computations

Example:

- Consider the sequence of all prime numbers:
2, 3, 5, 7, 11, 13, 17, 19, 23, ...
- the first 5 are 2, 3, 5, 7, 11

Sieve of Eratosthenes

The computation of the value of `e` can be delayed by "packing" it into a function (a **closure**):

```
fun () -> e
```

Example:

```
fun () -> 3+4;;  
val it : unit -> int = <fun:clo@10-2>  
  
it();;  
val it : int = 7
```

The addition is deferred until the closure is applied.

Example continued

One can make it visible when computations are performed by use of side effects:

```
let idWithPrint i = let _ = printfn "%d" i
                    i;;
val idWithPrint : int -> int

idWithPrint 3;;
3
val it : int = 3
```

The value is printed before it is returned.

```
fun () -> (idWithPrint 3) + (idWithPrint 4);;
val it : unit -> int = <fun:clo@14-3>
```

Nothing is printed yet.

```
it();;
3
4
val it : int = 7
```

A lazy list or *sequence* in F# is a possibly infinite, ordered collection of elements, where the elements are computed *by demand* only.

A natural number sequence $0, 1, 2, \dots$ is created as follows:

```
let nat = Seq.initInfinite (fun i -> i);;
val nat : seq<int>
```

A `nat` element is computed by demand only:

```
let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>
```

```
Seq.nth 4 nat;;
4
val it : int = 4
```

Further examples

A sequence of even natural numbers is easily obtained:

```
let even = Seq.filter (fun n -> n%2=0) nat;;  
val even : seq<int>
```

```
Seq.toList(Seq.take 4 even);;
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
val it : int list = [0; 2; 4; 6]
```

Demanding the first 4 even numbers demands a computation of the first 7 natural numbers.

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
2
- next sequence 3, 5, 7, 9, 11, ...
select head (3), and remove multiples of 3 from the sequence
2, 3
- next sequence 5, 7, 11, 13, 17, ...
select head (5), and remove multiples of 5 from the sequence
2, 3, 5
- ⋮

Sieve of Eratosthenes in F# (I)

Remove multiples of *a* from sequence *sq*:

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq;;  
val sift : int -> seq<int> -> seq<int>
```

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

```
let rec sieve sq =  
    Seq.delay (fun () ->  
        let p = Seq.nth 0 sq  
        Seq.append  
            (Seq.singleton p)  
            (sieve(sift p (Seq.skip 1 sq)))));;  
val sieve : seq<int> -> seq<int>
```

- Delay is needed to avoid infinite recursion
- `Seq.append` is the sequence sibling to `@`
- `Seq.nth 0 sq` gives the head of `sq`
- `Seq.skip 1 sq` gives the tail of `sq`

Examples

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

```
let primes = sieve(Seq.initInfinite (fun n -> n+2));;  
val primes : seq<int>
```

```
let nthPrime n = Seq.nth n primes;;  
val nthPrime : int -> int
```

```
nthPrime 100;;  
val it : int = 547
```

Re-computation can be avoided by using cached sequences:

```
let primesCached = Seq.cache primes;;
```

```
let nthPrime' n = Seq.nth n primesCached;;  
val nthPrime' : int -> int
```

Computing the 700'th prime number takes about 8s; a subsequent computation of the 705'th is fast since that computation starts from the 700 prime number

Sieve of Eratosthenes using Sequence Expressions

Sequence expressions can be used for defining step-by-step generation of sequences.

The sieve of Eratosthenes:

```
let rec sieve sq =
  seq { let p = Seq.nth 0 sq
        yield p
        yield! sieve(sift p (Seq.skip 1 sq)) };;
val sieve : seq<int> -> seq<int>
```

- By construction lazy – no explicit `Seq.delay` is needed
- `yield x` adds the element `x` to the generated sequence
- `yield! sq` adds the sequence `sq` to the generated sequence
- `seqexp1`
`seqexp2` appends the sequence of `seqexp1` to that of `seqexp2`

Example: Catalogue search (I)

Extract (recursively) the sequence of all files in a directory:

```
open System.IO ;;

let rec allFiles dir =
  seq {yield! Directory.GetFiles dir
       yield! Seq.collect allFiles (Directory.GetDirectories dir)}
val allFiles : string -> seq<string>
```

where

`Seq.collect: ('a -> seq<'c>) -> seq<'a> -> seq<'c>`
 combines a 'map' and 'concatenate' functionality.

```
Directory.SetCurrentDirectory @"C:\mrh\Forskning\Cambridge\";;
let files = allFiles ".";;
val files : seq<string>

Seq.nth 100 files;;
val it : string = ".\BOOK\Satisfiability.fs"
```

Nothing is computed beyond element 100.

Example: Catalogue search (II)

We want to search for files with certain extensions, e.g. as follows:

```
let funFiles=Seq.cache (searchFiles (allFiles ".") ["fs";"fsi"]);;
val funFiles : seq<string * string * string>

Seq.nth 0 funFiles;;
val it : string * string * string= (".\", \"CatalogueSearch\", \"fs\")

Seq.nth 6 funFiles;;
val it : string * string * string = (\".\BOOQ\", \"Curve\", \"fsi\")

Seq.nth 11 funFiles;;
val it : string * string * string
    = (\".\BOOQ\", \"Satisfiability\", \"fs\")
```

- a sequence is chosen so that the search is terminated when the wanted file is found
- a cached sequence is chosen to avoid re-computation

Example: Catalogue search (III)

The search function can be declared using regular expressions:

```
open System.Text.RegularExpressions ;;

let rec searchFiles files exts =
    let reExts = List.foldBack (fun ext re -> ext+"|" + re) exts ""
    let re = Regex (@"\G(\S*\))( [^\s\]+)\.(" + reExts + ")$")
    seq {for fn in files do
        let m = re.Match fn
        if m.Success
        then let path = captureSingle m 1
             let name = captureSingle m 2
             let ext = captureSingle m 3
             yield (path, name, ext) };;
    val searchFiles : seq<string> -> string list
    -> seq<string * string * string>
```

- `reExts` is a regular expression matching the extensions
- The path matches the regular expression `\S*\`
- The file name matches the regular expression `[^\s\]+`
- The function `captureSingle` can extract captured strings

Summary

- Anonymous functions `fun () -> e` can be used to **delay the computation** of `e`.
- Possibly infinite sequences provide natural and useful abstractions
- The computation by demand only is convenient in many applications

It is occasionally efficient to be lazy.

The type `seq<'a>` is a synonym for the .NET type `IEnumerable<'a>`.

Any .NET type that implements this interface can be used as a sequence.

- Lists, arrays and databases, for example.