02157 Functional Programming

Sequences and Sequence Expressions

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

- Sequences and Sequence expressions

- Property-based testing
  - Rehearsal of trees
  - Examples with sequences
  - Properties, generators and Shrinkers
  - A look at computation expressions for generators resembles sequence expressions

Sequence: a possibly infinite, ordered collection of elements, where the elements are computed by demand only

- the sequence concept
- standard sequence functions – the `Seq` library
- sequence expressions – computation expressions used generate sequences in a step by step manner

Computation expressions: provide a mean to express specific kinds of computations where low-level details are hidden. See Chapter 12.

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 a *sequence*, 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 is used in computations

Example:

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

Sieve of Eratosthenes

Delayed computations in eager languages

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.

How can we convince ourselves that the addition is deferred?

Example continued

A use of side effects may reveal when computations are performed:

```
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
```

Sequences in F#

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

The natural number sequence $0, 1, 2, \ldots$ is created as follows:

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

where `id:'a->'a` is the built-in identity function, i.e. $\texttt{id}(x) = x$

No element in the sequence is generated yet!

The type `seq<'a>` is an abstract datatype.

Programs on sequences are constructed from `Seq`-library functions

Explicit sequences and conversions for finite sequences

Two conversion functions

```
Seq.toList: seq<'a> -> 'a list

Seq.ofList: 'a list -> seq<'a>
```

with examples

```
let sq = Seq.ofList ['a' .. 'f'];;
val sq : seq<char> = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']

let cs = Seq.toList sq;;
val cs : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
```

Alternatively, a finite sequence can written as follows:

```
let sq = seq ['a' .. 'f'];;
val sq : seq<char> = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
```

Notice

- Seq.toList – does not terminate for infinite sequences
- seq $[x_1; \ldots; x_n]$ is a finite sequence with $n \geq 0$ elements

Selected functions from the library: Seq

- `initInfinite: (int ->'a) -> seq<'a>`.
  `initInfinite` f generates the sequence $f(0), f(1), f(2), \ldots$
- `delay: (unit->seq<'a>) -> seq<'a>`.
  `delay` g generates the elements of $g()$ lazily
- `collect: ('a->seq<'b>) -> seq<'a> -> seq<'b>`.
  `collect` f sq generates the sequence obtained by appending the sequences: $f(sq_0), f(sq_1), f(sq_2), \ldots$

The Seq library contains functions, e.g. `collect`, that are sequence variants of functions from the List library. Other examples are:

- `item: int -> seq<'a> -> 'a`
- `head: seq<'a> -> 'a`
- `tail: seq<'a> -> seq<'a>`
- `append: seq<'a> -> seq<'a> -> seq<'a>`
- `take: int -> seq<'a> -> seq<'a>`
- `filter: ('a->bool) -> seq<'a> -> seq<'b>`.

Example continued

A `nat` element is computed by demand only:

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

— using `idWithPrint` to inspect element generation.

Demanding an element of the sequence:

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

Just the 5th element is generated

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 requires a computation of the first 7 natural numbers.

Sieve of Eratosthenes

Greek mathematician (194 – 176 BC)

Computation of prime numbers

- start with the sequence $2, 3, 4, 5, 6, \ldots$
  select head (2), and remove multiples of 2 from the sequence
  2
- next sequence $3, 5, 7, 9, 11, \ldots$
  select head (3), and remove multiples of 3 from the sequence
  $2, 3$
- next sequence $5, 7, 11, 13, 17, \ldots$
  select head (5), and remove multiples of 5 from the sequence
  $2, 3, 5$
- $\vdots$

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.head sq
                  Seq.append
                      (seq [p])
                      (sieve(sift p (Seq.tail sq))));;

val sieve : seq<int> -> seq<int>
```

- A delay is needed to avoid infinite recursion Why?

Sequence expressions support a more natural formulation

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.item 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.item n primesCached;;
val nthPrime' : int -> int
```

Computing the 700'th prime number takes about 4.5s; 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 sequences in a step-by-step generation manner.

The sieve of Erastothenes:

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

- By construction lazy – no need to use `Seq.delay`
- `yield` *x* adds the element *x* to the generated sequence
- `yield!` *sq* adds the sequence *sq* to the generated sequence
- $seqexp_1$
$seqexp_2$ appends the sequences $seqexp_1$ and $seqexp_2$

Defining sift using Sequence Expressions

The sift function can be defined using an iteration:

`for` *pat* `in` *exp* `do` *seqexp*

and a filter:

`if` *exp* `then` *seqexp*

as follows:

```
let sift a sq = seq { for n in sq do
                          if n % a <> 0 then
                              yield n       };;
val sift : int -> seq<int> -> seq<int>
```

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

Nothing is computed beyond element 100.

Summary

- Anonymous functions `fun () -> ` *e* can be used to delay the computation of *e*.
- Possibly infinite sequences provide natural and useful abstractions
- Functions from the Seq-library
- Sequence expressions – step-wise sequence generation

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 and arrays, for example.

Property-based testing

- Rehearsal of trees
- Examples with sequences
- Properties, generators and Shrinkers
- A look at computation expressions for generators resembles sequence expressions

Consult `https://fscheck.github.io/FsCheck/` concerning installation and resources

An example

Consider the expressions:
```
let x = y + z + 3
in x + y + 4
```

The following expressions have the same value when y = 1 and z = 2:

```
6 + y + 4
substitute 6 for x
in x + y + 4
```

```
y + z + 3 + y + 4
substitute y + z + 3 for x
in x + y + 4
```

Suppose we have functions:

- `eval: E->Env->int` that evaluates *e* in environment *m*
- `subst: string->E->E->E` that performs substitutions

The property substitution preserves meaning

```
let substOK1(x, e1, e2, m) = //  "let x = e1 in e2 , m"
   let v1 = eval e1 m
   let e2' = subst x (C v1) e2
   let e2'' = subst x e1 e2
   eval e2'' m = eval e2' m;;
```

should hold for all expressions and environments that fit together.

Scenarios for substitution

Remember: `subst` x e_1 e_2 substitutes e_1 for x in e_2

Example 1:

```
let x = y+z                "subst x (y+z) (x+let x=6 in x)"
in x+let x=6 in x
```

- Only free occurrences of x in e_2 should be substituted

```
"subst x (y+z) (x+let x=6 in x)" = "y+z + let x=6 in x)"
```

Example 2:

```
let x=y                      "subst x y (let y=6 in x+y)"
in let y=6 in x+y
```

- Avoid that a free variable of e_1 is captured by a let-binding in e_2. Rename bound variables.

```
"subst x y (let y=6 in x+y)" = "let y1=6 in y+y1"
```

where `y1` is a "new" variable

Test and Verification

Tests:

- easy to write
- can reveal errors
- show correctness of a very limited number of concrete cases

low level of abstraction

Verification:

- complicated to complete
- provide guarantees
- focus of correctness properties

high level of abstraction

Between Test and Verification

Tests: ...

Property-based testing

- focus on properties of programs
- construction of programs
- a randomly generated sample covers edge cases and typical situations
- Short counter-examples are useful
- gives high confidence
- limited effort

enhances understanding

high level of abstraction

Verification: ...

Correctness properties are Boolean-valued functions

A property is a Boolean-valued function with type

$$\tau_1 \to \cdots \to \tau_n \to \texttt{bool}$$

Append is associative:

```
let appendAssocProp xs ys zs =
xs @ (ys @ zs) = (xs @ ys) @ zs
'a list -> 'a list -> 'a list -> bool when 'a : equality
```

The associative law can be tested on some examples:

```
let test = appendAssocProp [1;2] [3;4;5] [6;7;8;9];;
```

- But it requires discipline to come out with a suitable test suite

How can we get confidence in tests that should validate that

```
appendAssocProp xs ys zs
```

holds for all lists *xs*, *ys* and *zs*?

Property-based testing

Given

- property F with type $\tau_1 \to \cdots \to \tau_n \to \texttt{bool}$
- with input variable $x_1, \ldots, x_n$

the library FsCheck supports

- generation of random values for $x_1, \ldots, x_n$
- test whether F holds for all sample values
- presentation of a short counter-example when F is falsified

where x_i can have any monomorphic type.

```
open FsCheck

let appendAssocProp (xs: int list) ys zs =
xs @ (ys @ zs) = (xs @ ys) @ zs;;

Check.Quick appendAssocProp;;
Ok, passed 100 tests.
val it : unit = ()
```

PBT of substitution

```
#r "nuget: FsCheck";; open FsCheck;;
...
let substOK1(x, e1, e2, m) = //  "let x = e1 in e2 , m"
   let v1 = eval e1 m
   let e2' = subst x (C v1) e2
   let e2'' = subst x e1 e2
   eval e2'' m = eval e2' m;;

Check.Quick substOK1;;
(* ...
Original:
(null, C 0, V "", map [])
with exception:
System.Collections.Generic.KeyNotFoundException: The given
... *)
```

- Necessary: Values for *x*, *e*1, *e*2, *m* must fit together
- Convenient: “natural” variable names

Custom generators are needed

The types `Arbitrary<'a>` and `Gen<'a>`

An `Arbitrary<'a>` instance comprises a

- a generator *g* of type `Gen<'a>` and
- a shrinker *f* of type `'a -> seq<'a>`

to be used when testing properties.

Good default implementation exists for most types.

A generator *g* : `Gen<'a>` can be considered

- a computation of a random value of type `'a`

A shrinker *f* is a function that, for a given counter example, returns a sequence of smaller/simpler values.

The module Gen can be considered a library for forming new generators

New generators can also be formed using F#'s computation expressions

`Gen<'a>` is a monad using Haskell terminology

Selected pre-defined generators

```
Gen.constant: 'a -> Gen<'a>

Gen.oneof: seq<Gen<'a>> -> Gen<'a>

Gen.map2: ('a->'b->'c) -> Gen<'a> -> Gen<'b> -> Gen<'c>

Gen.sized: (int -> Gen<'a>) -> Gen<'a>
```

Some examples

```
> let g10 = seq {for i in 1 ..10 do
                   yield Gen.constant i};;
val g10: Gen<int> seq

> let g = Gen.oneof g10;;
val g: Gen<int> = Gen <fun:Bind@88>

> Gen.sample 1 5 g;;
val it: int list = [9; 3; 9; 3; 4]
> Gen.sample 1 5 g;;
val it: int list = [2; 7; 5; 4; 7]
```

Generators for substitution property

Two mail generators

```
myEnvGen: string list -> Gen<Env>

myEGen: string list -> Gen<E>
```

where

- `myEGen` *vs* generates an expression where only variables in *vs* can occur free.
- `myEnvGen` *vs* generates an environment for variables *vs*

Putting the pieces together:

```
type Subst = string * E * E * Env

mySubstGen: Gen<Subst>
```

where `mySubstGen` generates a tuple (x, e_1, e_2, m) satisfying

$$dom\ m = free\ e_1 \cup free\ e_2$$

Property `substOK1((x,e1,e2,m):Subst)` can now be checked.

Let us have a look at the programs

Further properties

where

- `myEGen` *vs* generates an expression where only variables in *vs* can occur free.
- `myEnvGen` *vs* generates an environment for variables *vs*

Putting the pieces together:

```
type Subst = string * E * E * Env

mySubstGen: Gen<Subst>
```

where `mySubstGen` generates a tuple (x, e_1, e_2, m) satisfying

$$dom\ m = free\ e_1 \cup free\ e_2$$

Property `substOK1((x,e1,e2,m):Subst)` can now be checked.

Let us have a look at the programs

Further properties

```
let rec elimLet e =
   match e with
   | Add(e1,e2)    -> Add(elimLet e1, elimLet e2)
   | Let(x,e1,e2) -> elimLet (subst x e1 e2)
   | e             -> e;;

let elimLetOK1((e,m): EwithEnv) =
       eval e m = eval (elimLet e) m;;

Check.Quick elimLetOK1;;


// substEnv: Env -> E -> E
let substEnv m e =
   Map.foldBack subst (Map.map (fun _ n -> C n) m) e;;

let substOK2((e,m): EwithEnv) =
   eval e m = eval (substEnv m e) Map.empty;;

Check.Quick substOK2;;
```

PBT is a useful technique for software validation

Fundamental properties of programs can be validated
— provided they can be implemented.

Libraries supporting PBT exist for many languages:
`https://en.wikipedia.org/wiki/QuickCheck`.