Sequence: Simple and efficient iterators

Simon Cruanes

July 8th, 2014

$ whoami

PhD student in Deducteam

Topic: *Automated Theorem Proving*

- In a nutshell: try to solve the unsolvable (Gödel, etc.)
- Symbolic computations
- Lots of data structures and algorithms

Iterators

- abstraction over iteration (enumerating values)
- present in many languages
- Java, C++, python, rust, C#, lua, etc.
- sometimes *built-in* syntax (python, java,...)
- OCaml: fold/iter higher-order functions more common

Still, would be useful in OCaml

- Conversion between containers: n^2 functions to write
  $\rightarrow$ in practice, at best `to_list` and `of_list`
- Missing functions (`Queue.mem`, `Array.for_all`, etc.)
- `flat_map: ('a -> 'b t) -> 'a t -> 'b t` inefficient on most containers
- combinators (`map`, etc.): eager, build intermediate structures

Solution

$\rightarrow$ we define a type `'a Sequence.t`

- lazy (possibly infinite)
- no intermediate structure
- efficient

Replace the for loop

OCaml's `for` loop is limited. Instead:

```
# Sequence.(1 -- 10_000_000 |> fold (+) 0);;
- : int = 50000005000000

# let p x = x mod 5 = 0 in
  Sequence.(1 -- 5_000
    |> filter p
    |> map (fun x-> x*x)
    |> fold (+) 0
  );;
- : int = 8345837500
```

One recursive function to write them all

```
type term = Var of string
          | App of term * term
          | Lambda of term ;;
let subterms : term -> term sequence = ...
```

Now we can define many other functions easily!

```
# let vars t =
    S.filter_map
        (function Var s -> Some s | _ -> None)
        (subterms t) ;;
val vars : term -> string sequence = <fun>

# let size t = Sequence.length (subterms t) ;;
val size : term -> int = <fun>

# let vars_list l = S.of_list l |> S.flat_map vars;;
val vars_list : term list -> string sequence = <fun>
```

Containers

```
# let contains_value x h =
    S.hashtbl_values h
    |> S.mem x ;;
- : 'b -> ('a,'b) Hashtbl.t -> bool

# let rev_tbl h =
    S.of_hashtbl h
    |> S.map (fun (x,y) -> y,x)
    |> S.to_hashtbl ;;
- : ('a,'b) Hashtbl.t -> ('b,'a) Hashtbl.t

# let tbl_of_list l = S.to_hashtbl (S.of_list l);;
- : ('a * 'b) list -> ('a,'b) Hashtbl.t

# let tbl_values h = S.to_list (S.hashtbl_values h) ;;
- : ('a, 'b) Hashtbl.t -> 'b list
```

Playing with Hashtbl

```
# let tbl = Sequence.(1 -- 1000
    |> map (fun i -> i, string_of_int i)
    |> to_hashtbl
    );;
- : (int, string) Hashtbl.t = <abstr>

# Hashtbl.length tbl;;
- : int = 1000

# Sequence.(hashtbl_keys tbl
    |> take 15
    |> iter (Hashtbl.remove tbl)
    );;
- : unit = ()

# Hashtbl.length tbl;;
- : int = 985
```

Combinatorics

Quite easy to backtrack using Sequence (+ early exit, fold...)

Example: Permutations of lists

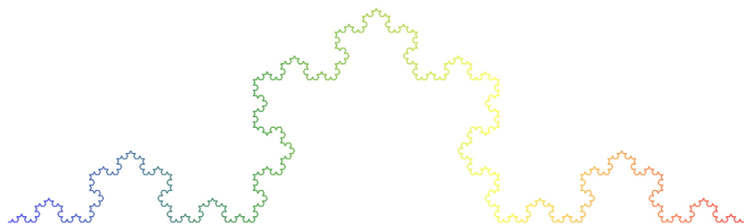
```
# module S = Sequence ;;
# let rec insert x l = match l with
    | [] -> S.return [x]
    | y::tl ->
        S.append
            S.(insert x tl >|= fun tl' -> y::tl')
            (S.return (x::l)) ;;

# let rec permute l = match l with
    | [] -> S.return []
    | x::tl -> permute tl >>= insert x ;;

# permute [1;2;3;4] |> S.take 2 |> S.to_list ;;
- : int list list = [[4; 3; 2; 1]; [4; 3; 1; 2]]
```

L-systems

Gabriel Radanne (@Drup): https://github.com/Drup/LILiS



Nested `flat_map` (convert segment into sub-segments)
(`flat_map : ('a -> 'b t) -> 'a t -> 'b t`)

Currently in OCaml

- standard library: `Stream.t` (slow, designed for IO)
- Batteries has `Enum.t` (slow, complicated)
- Core: very recently, `core.sequence` (requires Core)

→ roll my own iterators (fast, self-contained)

Survey: Possible Implementations

Roughly

```
type 'a gen = unit -> 'a option;;

type 'a BatGen.t = unit -> 'a node
and 'a node =
| Nil
| Cons of 'a * 'a BatGen.t ;;

type 'a sequence = ('a -> unit) -> unit ;;
```

- Possibility to use structural types
- Possibility to use exceptions for end-of-iterator
- Monadic versions (`Lwt_stream.t`)

Sequence

Choose `'a sequence = ('a -> unit) -> unit`:

- Simple
- Very efficient
- Structural type (interoperability!)
- Easy to define on opaque types (if `iter` provided)
  $\rightarrow$ definable on `Set.S.t`, `Queue.t`, `Hashtbl.t`, etc.
  $\rightarrow$ good for interoperability
- Expressiveness: "good enough" (more details later)

Benchmarks (L-systems)

```
--- Lsystem Von_koch for 7 iterations ---
              Rate        Stream
  Stream 2.91+-0.02/s         --
    Enum 13.5+- 0.3/s       362%
     Gen 36.4+- 0.0/s      1150%
  BatSeq 42.8+- 0.2/s      1369%
Sequence 51.4+- 0.1/s      1664%

--- Lsystem dragon for 15 iterations ---
              Rate        Stream
  Stream 1.81+-0.00/s         --
    Enum 9.70+-0.12/s       436%
     Gen 22.4+- 0.1/s      1140%
  BatSeq 26.2+- 0.1/s      1349%
Sequence 34.8+- 0.1/s      1823%
```

Credits to @Drup. This benchmarks mostly `flat_map`.

Limitations

Sequence isn't perfect:

- Some operators impossible to write
  - → `combine`, `sorted_merge`, etc.
  - → other iterators can do it (`opam install gen`)
  - → possible with `delimcc`
  - → possible with `Sequence.persistent` (store into list)
- meh for IO
  - → would need a monad (Lwt/Async)
  - → resource handling
  - → other iterators: same problems

The Sequence Library

- BSD-licensed
- Provides many combinators and conversion functions
- Package sequence on opam

Implementation

quite easy: call continuation k to yield an element

```
let map f seq = fun k -> seq (fun x -> k (f x));;

let flat_map f seq = fun k -> seq (fun x -> (f x) k);;

let filter p seq = fun k -> seq (fun x -> if p x then k

let iter f seq = seq f ;;

let of_list l = fun k -> List.iter k l
```

Conclusion

- Efficient, simple, lazy, *structural* iterators
- Used a lot in my code
    - backtracking algorithms (n-ary unification)
    - traversing nested structures
    - missing `for_all`, `flat_map`, `filter_map`, ...operators
    - ...
- Works on opaque (third-party) containers
- Free software

Questions?

?