DE GRUYTER OPEN

JACSM 2014, Vol. 6, No. 1, pp. 67 - 80

DOI 10.2478/jacsm-2014-0005

PERSISTENT SEQUENCES WITH EFFECTIVE RANDOM ACCESS AND SUPPORT FOR INFINITY

Konrad Grzanek

IT Institute, University of Social Sciences, Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Persistent sequences are the core data structure in functional programming style. Their typical implementations usually allow creating infinite streams of objects. Unfortunately, asking for length of an infinite data structure never ends or ends with a run-time error. Similarly, there is no default way to make an effective, *O*[*1*] or logarithmic access to an arbitrarily chosen sequence element, even when the nature of the correlation between index value and the sequence element is known. This paper presents a Clojure library that meets these limitations and offers an enhanced version of sequences with a support for effective random access and the ability to ask for an infinite length.

Key words: functional programming, Clojure, persistent collections, infinity, random access

1 Prerequisites for Enhanced Sequences

Sequences are by far the most common data structures in functional programming languages [3]. The name *Lisp* (originally *LISP*) derives from *LISt Processing*. In *Haskell* [5], [6] the list structure is also a basic non-atomic data type. Sequences possess a set of useful algebraic properties; they are *functors* and *monadic type constructors* [8]. Modern programming languages like *Clojure* [1], [2] and *Haskell* support non-strict evaluation This property of these languages makes creating infinite sequences natural. For instance in Clojure the expression

(***iterate*** *inc 0*)

and in Haskell, an equivalent form

iterate (1+) 0

of type

Num a ⇒ [*a*]

both denote an infinite sequence of natural numbers *0, 1, 2, 3, ...* In Clojure the function *clojure.core/iterate* is defined like

```
(defn iterate
  [f x] (cons x (lazy-seq (iterate f (f x)))))
```

and in Haskell

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

Moreover using *co-recursion* (see e. g. [4])the definitions like the infinite stream of values (*1* in this case)

ones = 1 : ones

or odd numbers

odds = 1 : map (+2) odds

are perfectly possible. In Clojure the lazy evaluation is not a default behavior, but the two co-recursive Haskell streams may be expressed like

```
(def ones (cons 1 (lazy-seq ones)))
```

and

```
(def odds (cons 1 (lazy-seq (map #(+ 2 %) odds))))
```

There is even a possibility to build a generalization of the emerging pattern using Clojure *macro*:

```
(defmacro lazy-cons
  [x seq]
  `(cons ~x (lazy-seq ~seq)))
```

and then one can write the expressions as
(***def*** *ones* (***lazy-cons*** *1 ones*))

(***def*** *odds* (***lazy-cons*** *1* (***map*** #(+ *2 %*) *odds*)))

This mimics the Haskell expressions closely, the differences are more on the syntactic than the semantic level with one clear semantic mismatch: the (:) operator in Haskell is a function of type $a \rightarrow [a] \rightarrow [a]$ and the *lazy-cons* introduced by us in Clojure is a macro that – as such – can't be passed around as a *first-class* value. This is still not an issue when defining streams co-recursively.

Infinite sequences (streams) are first-class citizens in the functional programming style under an assumption of non-strict (presumably lazy) evaluation. Although the examples given above make it clear, some questions still arise about querying these sequences for their length or about accessing their arbitrarily chosen element.

Let's introduce the symbol $\perp$ to depict a polymorphic "result" of a never ending computation and let the symbol $\Rightarrow$ mean "evaluates to" or "reduces to". The following are true

length ones $\Rightarrow \perp$

and

(*count odds*) $\Rightarrow \perp$

This is almost never a desirable computer program behavior. When we assume the infinity of some streams, it is natural to expect

length ones $\Rightarrow \infty$

and

(*count odds*) $\Rightarrow \infty$

Moreover, even in the face of operating on finite sub-streams, it is impossible to write

(*count* (*take* (** *2 1000*) *ones*))

and get a correct result, because

```
(defn count
  [coll] (clojure.lang.RT/count coll))
```

and

```
public static int count(Object o){
      if(o instanceof Counted)
            return ((Counted) o).count();
      return countFrom(Util.ret1(o, o = null));
}
```

The *count* method returns *int* values and so it can't give a *long* or (in this case) a *java.math.BigInteger* answer. In Haskell the function *length* is of type $[a] \rightarrow Int$, so it suffers the same kind of problems.

Another variant of these stream library inconveniences occurs with respect to a random access. In Clojure the *clojue.core/nth* procedure requires an *int* and in Haskell the operator (!!) has type $[a] \rightarrow Int \rightarrow a$. Moreover, the realizations of these operators in both languages take the same approach, like in the Haskell code below:

```
xs !! n | n < 0 = error ...
[] !! _         = error …
(x : _) !! 0    = x
(_ : xs) !! n   = xs !! (n-1)
```

This is why it is impossible to introduce some more effective ways to access the nth element in the sequence.

We aim here to present a Clojure library enhancement that solves the problems specified above. A similar, but not identical solution may be proposed for Haskell. The differences are caused by Haskell's static type checking and lack of sub-typing[1] ([6]).

2 Enhanced Sequences Implementation and Usage

The solution consists of a new sequence type that implements the following interface:

1 The Haskell solution requires introducing a specialized algebraic data-type with all interesting list operators, including the effective ones proposed in this paper, defined for it. Although it may seem disturbing at first, it is rather natural regarding the nature of Haskell type system.

```
public interface IEnhancedSeq extends
      IPersistentCollection, Sequential,
      Indexed, IHashEq {

  Number len();

  Object enth(Number n);
}
```

An enhanced sequence's length is represented here by a *len* operator of type *IEnhancedSeq → java.lang.Number*. This opens a way to return length values greater than *Integer.MAX_VALUE*. Similarly, the *enth* operator of type *IEnhancedSeq → java.lang.Number → Object* allows using *Numbers* instead of *int* values as indexes when accessing arbitrary sequence elements.

The enhanced sequences type implements Clojure interfaces common for predefined kinds of sequential types in the language, as presented at the above *IEnhancedSeq* interface definition.

The interfaces are implemented by a single abstract class *ESeq*. This section gives a detailed description of the class. The class definition goes as follows:

```
public abstract class ESeq implements List, IEnhancedSeq
{
  private final IPersistentCollection origin;
}
```

There are two constructor methods in the class. The one called *withLen* allows to bind a length generating procedure to the resulting enhanced sequence:

```
  public static Object withLen(final IFn len,
                               Object coll) {
    return new ESeq(origin(coll)) {
      @Override
      public Number len() {
        return (Number) len.invoke();
      }
    };
  }
```

The other one – *withEnth* – binds a random accessor:

```
public static Object withEnth(final IFn enth,
                              Object coll) {
    return new ESeq(origin(coll)) {
      @Override
```

```
      public Object enth(Number n) {
        return enth.invoke(n);
      }
    };
  }
```

After the enhanced objects are created they may be asked either for their length:

```
  @Override
  public Number len() {
    if (origin instanceof IEnhancedSeq) {
      return ((IEnhancedSeq) origin).len();
    }
    return origin.count();
  }
```

or for their nth element:

```
  @Override
  public Object enth(Number n) {
    if (origin instanceof IEnhancedSeq) {
      return ((IEnhancedSeq) origin).enth(n);
    }
    return RT.nth(origin, RT.intCast(n));
  }
```

In the two operators above a wrapped origin collection is used. In fact an *ESeq* is just a wrapper around the origin, one may say – a decorator – that offers the additional useful bindings established on the compile time.

The implementation works smoothly with the standard library mechanisms, because the original operators simply use the newly introduced enhancements, like

```
  @Override
  public final int count() {
    return RT.intCast(len());
  }
```

or

```
  @Override
  public final Object nth(int i) {
    return enth(i);
  }
```

and it's variant:

```
@Override
public final Object nth(int i, Object notFound) {
  try {
    return enth(i);
  }
  catch (IndexOutOfBoundsException e) {
    return notFound;
  }
}
```

An interesting point to be made here is the implementation of the operator that checks for the enhanced sequence emptiness. It uses a special *Infinity* type[2]. As it can be seen at the following listing, the *Infinity.POSITIVE* is an instance of *java.lang.Number* and it may be returned by the *len* operator:

```
private static final Long ZERO = 0L;

@Override
public final boolean isEmpty() {
  Number n = len();
  if (Infinity.POSITIVE == n) {
    return false;
  }
  return Numbers.equiv(n, ZERO);
}
```

This closes up the implementation details of *ESeq* class. There are many more mechanisms that have their place in the whole, but because their role is limited to ensuring the conformance with the standard library and it's contracts, we decided not to present them here.

On Clojure side there is one basic operator which answers whether or not the passed object (presumably a collection) is an enhanced seq or not:

```
(defn eseq?
  [coll]
  (instance? kongra.core.eseq.IEnhancedSeq coll))
```

2 Presenting the details of the *Infinity* type including the basic infinity-aware arithmetic operators lay beyond the scope of this paper.

2.1 Length and Infinity

The mechanisms described in the previous section are used to build another layer of abstraction, the one containing enhanced length-related operators. The most basic one allows to bind length generating function to the returned sequence (as mentioned above).

```
(defn with-len
  [len coll]
  (kongra.core.eseq.ESeq/withLen len coll))
```

There is also a possibility to pass a length value that gets bound immediately. The expression (*return* <*value*>) generates a function that always returns <*value*> and can be used as an argument to *with-len*:

```
(defn with-len-value
  [value coll]
  (with-len (return value) coll))
```

A useful macro *with-delayed-len* allows one to bind a lazily-evaluated length value:

```
(defmacro with-delayed-len
  [len-expr coll]
  `(let [d# (delay ~len-expr)]
     (with-len (return @d#) ~coll)))
```

and the operator *with-len-like* copies the binding for length from the origin into the resulting collection:

```
(defn with-len-like
  [origin coll]
  (with-len-value (len origin) coll))
```

Finally one can ask the collection for it's enhanced length value using the following *len* operator. It is worth noting that for collections of types other than *ESeq* simply *clojure.core/count* is used to establish the value:

```
(defn len
  [coll]
  (if (eseq? coll)
    (.len ^kongra.core.eseq.IEnhancedSeq coll)
    (count coll)))
```

As we stated in the previous section, the *len* operator may return the Infinity value. As a natural and desired consequence of this fact we may ask the collection if it is infinite or not. The following procedures simply compare the returned length value with Infinity constants:

```
(defn infinite?
  [coll]
  (+∞? (len coll)))


(defn finite?
  [coll]
  (not (infinite? coll)))
```

Two things should be underlined here:

1. If the collection asked for it is infinite length is not an enhanced one with len bound, then the count operator is used. This standard library function simply counts the sequence elements traversing it from the start. So the len operator is O[n] in the worst case.
2. No standard Clojure collection is an ESeq by default. This is why both hold: (infinite? (iterate inc 0)) ⇒ ⊥ and (infinite? odds) ⇒ ⊥

It is programmer's responsibility to mark a sequence as an infinite one. Fortunately, whe following simple procedure does the job.

```
(defn infinite
  [coll]
  (with-len-value +∞ coll))
```

Now with the following definitions

(***def*** *ones* (***infinite*** (***lazy-cons*** *1 ones*)))

(***def*** *odds* (***infinite*** (***lazy-cons*** *1* (***map*** #(+ *2 %*) *odds*))))

we have (*infinite? ones*) ⇒ *true*, (*infinite? odds*) ⇒ *true* and also (*len ones*) ⇒ ∞, (*len odds*) ⇒ ∞.

2.2 Random Access with Optimistic Performance Profile

Similarly, there are two operators, one that allows binding the effective random accessor procedure to the returned collection:

```
(defn with-enth
```

```
  [nth coll]
  (kongra.core.eseq.ESeq/withEnth nth coll))
```

and another that actually makes the access using the bound accessor, if present. In the face of absence of such an accessor binding, the standard *clojure.core/nth* is used:

```
(defn enth
  [coll n]
  (if (eseq? coll)
    (.enth ^kongra.core.eseq.IEnhancedSeq coll n)

    (nth coll n)))
```

3 Examples

To get a stronger grasp on what these enhancements may be useful for, please, take a look at the following procedure that concatenates collections and is aware of the possible infinity of some of the arguments:

```
(defn cat-eseq
  "Returns an enhaced collection being the result of concatenating the
  passed colls. It is assumed the number of colls is finite."
  [& colls]
  (let [;; prepare colls and lens (delayed)
        len-colls-bundle
        (delay
          (let [lens (map len colls)
                ;; take only the colls up to the first with
                ;; len=+∞ (inclusively)
                ∞-idx (find-idx' +∞? lens)
                colls (if ∞-idx (take (inc' ∞-idx) colls) colls)
                lens  (if ∞-idx (take (inc' ∞-idx) lens)  lens)]
            (pair lens colls)))

        lens  #(pair-first  @len-colls-bundle)
        colls #(pair-second @len-colls-bundle)

        ;; prepare enth (with delayed realization)
        intervs-intermap-bundle
        (delay
          (let [intervs (->> (lens)
                             (cons 0)
                             (apply cummulative-intervs')
                             vec) ;; essential wrt performance
                intermap (zipmap intervs (colls))]
            (pair intervs intermap)))

        intervs  #(pair-first  @intervs-intermap-bundle)
        intermap #(pair-second @intervs-intermap-bundle)

        enth-impl
```

```
        (fn [n]
          ;; 1. select proper interval
          (let [intv (binary-search (intervs) n
                                    #(interv-compare (lv "[, )") %2 %1))]
            (when-not intv (terror IndexOutOfBoundsException n))

            (let [ ;; 2. select collection
                  coll ((intermap) intv)
                  ;; 3. calculate the index in the collection
                  i (- n (:start intv))]
              ;; 4. get the result wrapped with the transformation
              (enth coll i)))))]

    (->> (apply concat (colls))
         (with-enth enth-impl)
         (with-delayed-len (reduce ∞+' (lens))))))
```

A slightly less complicated is the sequence of natural numbers. The first variant is not an enhanced one:

```
(defn N'
  "Returns an infinite N (natural numbers) set := 0, 1, 2, ...
  Optionally allows to specify the start (first) number. Unlimited
  integral range."
  ([]      (N' 0))
  ([start] (iterate inc' start)))
```

and now the enhanced version:

```
(defn N'-eseq
  ([] (N'-eseq 0))

  ([start]
     (->> (N' start)
          infinite
          (with-enth #(do
                        (when (< % 0) (terror
                           IndexOutOfBoundsException %))
                        (+' start %))))))
```

Similarly, the infinite sequence of factorial numbers, non-enhanced in the first place:

```
(defn factorials'
  "Returns an infinite stream 0!, 1!, 2!, 3!, ..."
  []
  (iterate-with *' 1 (N' 1)))
```

and it's enhanced version:

```
(defn factorials'-eseq
  "An eseq version of factorials'"
  []
  (->> (factorials')
       (with-enth #(factorial' %))
       infinite))
```

The *power-set* (set of all subsets) implementation takes a slightly more composite approach. First, the generator:

```
(defn- powerset-generator
  [indexed-coll n]
  (->> indexed-coll
       (take (ebit-length n))
       (filter (fn [p] (ebit-test n (pair-first p))))
       (map pair-second)))
```

then, the accessor

```
(defn nth-in-powerset
  "Returns an n-th element of a powerset of a collection. Works for
  n : T <: Long and for (possibly) infinite collections. That's why
  for the finite colls there are no range checks for n."
  [coll n]
  (powerset-generator (indexed' coll) n))
```

And the actual power-set sequence:

```
(defn powerset
  "Returns a powerset for a possibly infinite coll."
  [coll]
  (let [indexed-coll (indexed' coll)]
    (->> (N' 1)
         (map #(powerset-generator indexed-coll %))
         (take-while seq)
         (cons '()))))
```

The enhanced power-set sequence takes an additional element transformation function, called *enthtrans*, to (optionally) modify any value either when accessing it during a standard iteration (e.g. left or right *catamorphism* – see [7]) or by an accessor:

```
(defn powerset-eseq
  "Returns an enhanced version of the powerset. Every element of the
  returned collection is subjected to a transformation using enthtrans
  (identity by default)."
  ([enthtrans coll]
     (let [n (len coll)
           result (->> coll
                       powerset
                       (map enthtrans)
                       (with-enth #(enthtrans (
                                    nth-in-powerset coll %))))]
       (if (+∞? n)
         (infinite result)
         (with-delayed-len (** 2 n) result))))
  ([coll]
     (powerset-eseq clojure.core/identity ;; enthtrans
                    coll)))
```

The final case-study is an enhanced permutations generation routine. The accessor returns *n*-th permutation of a collection of elements using *Lehmer code* for *n* [9]:

```
(defn nth-permutation
  "Returns n-th permutation (lexicographical) of the given coll."
  [coll n]
  (let [v (if (vector? coll) coll (vec coll))
        lehmer-code (factoradic n)

        ;; lehmer-code must be supplemented with 0-s to match the
        ;; length of v
        zeros-count (- (count v) (count lehmer-code))
        lehmer-code (concat (repeat zeros-count 0) lehmer-code)

        gen (fn [[_ v] i] (pair (nth v i) (vec-remove i v)))]

    (->> (iterate-with gen (pair nil v) lehmer-code)
         next
         (map pair-first))))
```

The original lexicographical permutations are returned by *clojure.math.combinatorics/permutations* routine. This is wrapped within the following enhanced form:

```
(defn permutations-eseq
  "Returns an enhanced version of permutations. Every element of the
  returned collection is subjected to a transformation using enthtrans
  (identity by default)."
  ([enthtrans coll]
     (->> coll
          permutations
          (map enthtrans)
          (with-delayed-len (factorial' (len coll)))
          (with-enth #(enthtrans (nth-permutation coll %)))))

  ([coll] (permutations-eseq clojure.core/identity ;; enthtrans
                             coll)))
```

Here we also have an optional element transformation (*enthtrans*), like in the case of the power-set.

4 Conclusions

We presented a set of convenient extensions for the sequence abstraction in Clojure. The attached case studies show the relative ease of using these mechanisms. In general, the enhanced sequences fit pretty well in the ecosystem Clojure standard library. However, it would be an instructive experience to implement them in statically typed languages like Haskell. The Clojure solution is based on sub-typing, which is typical for a language that compiles down to Java byte-code and runs on top of the JVM. In Haskell there is no sub-typing, so the expected implementation technique presumably should consist of:

– using the algebraic data types
– using type-classes
– and finally – the resulting enhancement abstraction should make it's usage explicit rather than implicit as in the case of Clojure.

As a reward, one would get a statically typed, provably correct solution. It is now a question of undertaking future efforts to make this happen.

References

1. Halloway S., 2009, *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
2. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7
3. Bird R., Wadler P., *Introduction to Functional Programming*, 1988, Prentice Hall International (UK) Ltd
4. Doets K., van Eijck J., *The Haskell Road to Logic, Math and Programming*, 2004, College Publications, ISBN-10: 0954300696, ISBN-13: 978-0954300692
5. Lipovaca M., *Learn You a Haskell for Great Good*, 2011, ISBN: 978-1-59327-283-8
6. *Haskell Wikibook*, 2014, http://en.wikibooks.org/wiki/Haskell
7. Meijer E., Fokkinga M.M., *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*, 1991, Springer Verlag
8. Awodey S., *Category Theory, Second Edition*, 2010, Oxford University Press
9. Lehmer D.H., *Teaching combinatorial tricks to a computer*, 1960, Proc. Sympos. Appl. Math. Combinatorial Analysis, Amer. Math. Soc. 10: pp. 179–193