Towards the best collection API

A design of the overall optimal collection traversal interface

An argument against *iterator*

How to turn *any* enumerator inside out, into a stream

`http://pobox.com/~oleg/ftp/`
`papers/LL3-collections-enumerators.txt`

Terminology

Collection – a hash table; a file; a resultset; a generating function

Enumerator – a higher-order traversal function that applies a handler to each element
Synonyms: iterator (OCaml), for-each, fold

Cursor – an accessor of the current element, and, potentially, of the next one
Synonyms: iterator (C++), stream, lazy list

Conclusions

Enumerators should be offered *natively* in a collection API.
We can always *derive* cursors.

A generic procedure to turn any enumerator into a cursor, in a language with or without call/cc.

A procedure to turn an enumerator into a generator.

Cursors are useful and sometimes indispensable – *but not that often*. Why to use enumerators most of the time.

Enumerators vs. cursors (1/3)

Ease and safety of programming

- Enumerators are far easier to write: e.g., traversing a tree without parent pointers

- The current element of a cursor is an implicit state: cf. global variables

- Enumerators perfectly hide the traversal state

- Enumerators need no exceptions or out-of-band values to indicate the end of the traversal

Enumerators vs. cursors (2/3)

Efficiency

- A cursor must check for the validity of its state on each operation:
  `fgetc()` N times vs. `fread()` on the buffer of size N,
  nil checking in head/tail functions

- "The performance of cursors is horrible in almost all systems. One of us once had an experience of re-writing an eight-hour query having nested cursors into a cursor-free query that took 15 seconds." D. E. Shasha and P. Bonnet. DDJ, July 2002, pp. 46-54.

- Enumerators lend themselves to multi-stage programming.
  Inlining of iterations: Blitz++

Enumerators vs. cursors (3/3)

Predictable resource usage and avoidance of resource leaks

```
(define (enum proc collection)
  (let ((database-connection #f))
    (dynamic-wind
      (lambda ()
        (set! database-connection
          (open-connection collection)))
      (lambda () (iterate proc))
      (lambda ()
        (set! database-connection
          (close-connection database-connection))))))
```

No such simple bracketing for a cursor passed from one procedure to another.

In general, a cursor requires a manual resource management, or *finalization*.

Enumerators and Generators

Generator: an expression that can produce several values, on demand [Icon].

Generators share some advantages of enumerators and some drawbacks of cursors:

- Easy to write
- Good encapsulation of the traversal state
- Demand-driven: leak resources when the iteration is logically finished

Generators are trivial in Scheme: the first hint that enumerators and cursors are related via first-class continuations.

Multiple-valued expressions and shift/reset

Icon

```
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```

Scheme

```
(define (fail) (shift c "no"))   ; abort

(reset
  (let ((i (find "or"
                 "Store it in the neighboring harbor"))
    (if (not (> i 5)) (fail)
      (begin (display i) (newline)))))
```

Olivier Danvy and Andrzej Filinski: Abstracting Control. Proc. *1990* ACM Conf. on LISP and Functional Programming.

Generators in Python and Scheme

Python

```
>>>> # A recursive generator that generates Tree leaves in i
>>> def inorder(t):
...     if t:
...         for x in inorder(t.left):
...             yield x
...         yield t.label
...         for x in inorder(t.right):
...             yield x
```

Scheme

```
; tree:: '() | (list label tree-left tree-right)
(define (inorder tree)
  (lambda (suspend)
    (if (pair? tree)
        (apply
         (lambda (label left right)
           (ffor-each suspend (generative (inorder left)))
           (suspend label)
           (ffor-each suspend (generative (inorder right))))
         tree))))
```

`suspend` is an ordinary procedure

Complete code:

http://pobox.com/~oleg/ftp/Scheme/enumerators-callcc.html

Proposed traversal interface

The following interface ought to be provided *natively* by a collection API:

A left-fold enumerator with explicit multiple state variables and a premature termination.

In a language with call/cc:

```
coll-fold-left COLL PROC SEED ... -> [SEED ... ]

PROC VAL SEED ... -> [INDIC SEED ...]
```

In a language without call/cc:

```
coll-fold-left-non-rec COLL SELF PROC SEED ...
-> [SEED ... ]
```

Cursors are not banished and still available

enumerator ⟷ *stream*

We can always do stream → enumerator

The converse is true! For any collection, for any enumerator, we can invert an enumerator inside out and get a stream.

We conclude:

- Enumerators and streams are inter-convertible

- Native enumerators are a better API choice than native cursors

How to invert an enumerator

```
(define (lfold->lazy-list lfold collection)
  (delay
    (call-with-current-continuation
      (lambda (k-main)
        (lfold collection
          (lambda (val seed)
            (values
              (call-with-current-continuation
                (lambda (k-reenter)
                  (k-main
                    (cons val
                      (delay
                        (call-with-current-continuation
                          (lambda (k-new-main)
                            (set! k-main k-new-main)
                            (k-reenter #t))))))))
              seed))
          '())                              ; Initial seed
        (k-main '()))))))
```

From *any* left fold enumerator for *any* collection – into a stream

Inversion in a language with no call/cc (1/2)

The primitive construct is a non-recursive enumerator `CFoldLeft’`

```
  -- recursive enumerator
 type CFoldLeft coll val m seed =
        coll -> CollEnumerator val m seed
 type CollEnumerator val m seed =
        Iteratee val seed
        -> seed      -- the initial seed
        -> m seed
 type Iteratee val seed = seed -> val -> Either seed seed

  -- non-recursive enumerator
 type CFoldLeft’ val m seed =
        Self (Iteratee val seed) m seed
        -> CollEnumerator val m seed
 type Self iter m seed = iter -> seed -> m seed
 type CFoldLeft1Maker coll val m seed =
        coll -> m (CFoldLeft’ val m seed)
```

`CFoldLeft1Maker` should be offered natively by a collection API

Inversion in a language with no call/cc (2/2)

`CollEnumerator` is a fixpoint of `CFoldLeft'`

```
hfold_nonrec_to_rec:: (Monad m) =>
       coll -> (CFoldLeft1Maker coll val m seed)
            -> m (CollEnumerator val m seed)
hfold_nonrec_to_rec coll hfold1_maker = do
   hfold_left' <- hfold1_maker coll
   return $ fix hfold_left'
fix f = f g where g = f g
```

A stream is a continuation of `CFoldLeft'`

```
data MyStream m a = MyNil (Maybe a) |
                    MyCons a (m (MyStream m a))

hfold_nonrec_to_stream::
   (Monad m) => CFoldLeft' val m (MyStream m val)
                -> m (MyStream m val)
hfold_nonrec_to_stream hfold_left' = do
  let k fn (MyNil Nothing)  = return $ MyNil Nothing
      k fn (MyNil (Just c))
        = return $ MyCons c
                    (hfold_left' k fn (MyNil Nothing))
  hfold_left' k (\_ c -> Right $ MyNil $ Just c)
                (MyNil Nothing)
```

Note the polymorphic types!

In Practice

The enumerator `coll-fold-left` has been implemented, tested, and used:

- A relational database interface for Scheme, used in the production environment
- A Scheme TIFF image library
- Basic Linear Algebra and Optimization classlib (C++). Enumerator views of matrices added in Jan 1998.
- Under consideration for an Oracle RDBMS binding in Haskell