Bidirectional leveled enumerators

Irène Durand [1]

LaBRI, University of Bordeaux

April 27th, 2020

European Lisp Symposium, Zürich, Switzerland ELS2020

1. Joint work with Bruno Courcelle and Michael Raskin

Enumeration : what for ?

Enumeration is essential

- In general
  - for infinite sets of objects
  - for sets that are too big to be represented *in extenso*
    Python : `for i in range(n):`
- In particular
  - search problems with large answer sets
  - queries on large databases : iterators in `Java` and `SQL` :
  - constraint satisfaction problems

    Enumeration strategies for solving constraint satisfaction problems : A performance evaluation.
    Advances in Intelligent Systems and Computing, 347 :169–179, 07 2015.
  - ...

Framework

Development of the TRAG system : Term Rewriting Automata and Graphs

- `https://idurand@bitbucket.org/idurand/trag.git`
- core system entirely written in Common Lisp
- A web interface [2] : `trag.labri.fr` (uses some `JavaScript`)

- need for enumeration ⇒ `Enum` package, self-contained
- presented at ELS2012 in Zadar

2. developed by Michael Raskin

Overview of the talk

- Definition of an enumerator
- Presentation of the `Enum` package
- Problems raised by the enumeration of products
- Bidirectional leveled enumerators for enumerating products
- Conclusion

Definition of an enumerator

An enumerator E is a state machine which outputs the elements of a sequence $\widehat{E} = e_0, e_1, \ldots$ one at a time. The sequence may be finite $(e_n)_{n \in [0,c[}$ or infinite $(e_n)_{n \in \mathbb{N}}$.

Examples of sequences

1. the sequence of the days of the week :
   monday, tuesday, wednesday, thursday, friday, saturday, sunday
2. the sequence of all natural integers : $0, 1, 2, \cdots$
3. the alternating sequence : $1, -1, 1, -1, \cdots$
4. the sequence of prime numbers : $2, 3, 5, 7, 11, 13, \cdots$

Two basic operations :

- is there a next element ? (`next-element-p`)
- output the next element and move on (`next-element`)

API for Enumerator (basic operations)

```
(defclass abstract-enumerator () ())

(defgeneric next-element-p (enumerator)
  (:documentation "returns NIL if there is no next element,
   a non NIL value otherwise"))

(defgeneric next-element (enumerator)
  (:documentation "returns the next element,
   moves to the following one"))
```

Enumerator (main operation)

```
(defgeneric call-enumerator (enumerator)
  (:documentation
   "return as first value the next element of ENUMERATOR
    if it exists NIL otherwise
    as second value T if element was produced")
  (:method ((e abstract-enumerator))
    (if (next-element-p e)
        (values (next-element e) t)
        (values nil nil))))
```

Example of the list enumerator

```
ENUM> (setq *abc* (make-list-enumerator '(a b c)))
=> #<LIST-ENUMERATOR {100ADDAAC3}>
ENUM> (next-element *abc*) => A
ENUM> (next-element-p *abc*) => T
ENUM> (next-element *abc*) => B
ENUM> (call-enumerator *abc*)
C
T
ENUM> (call-enumerator *abc*)
NIL
NIL
ENUM> (collect-enum *abc*) => (A B C) ; only if finite
ENUM> (defparameter *ab-i* (make-list-enumerator '(a b) :circ t))
*I*
ENUM> (collect-n-enum *i* 10)
(A B A B A B A B A B)
```

Simple vs relying enumerators

- simple enumerators : do not rely on other enumerators
  - constant enumerator
  - enumerator of the elements of Lisp sequence (`list`, `vector`)
  - enumerator of a sequence defined inductively
  - ...
- relying enumerators : rely on other enumerators
  - `sequential`$(E^1, \ldots, E^n)$
  - `parallel`$(f, (E^1, \ldots, E^n))$
  - `filter`(p, E)
  - `product`$(f, (E^1, \ldots, E^n))$
  - ...

A relying enumerator : `parallel-enumerator`

The `parallel-enumerator` is like the enumerator version of the (`mapcar function list &rest more-lists`).

```
ENUM> (mapcar #'list
              '(e^1_0 e^1_1 e^1_2 e^1_3)
              '(e^2_0 e^2_1 e^2_2)
              '(e^3_0 e^3_1 e^3_2 e^3_3))
((E^1_0 E^2_0 E^3_0) (E^1_1 E^2_1 E^3_1) (E^1_2 E^2_2 E^3_2))
```

It stops with the shortest sequence.
Let $E = \texttt{parallel}(f, (E^1, \cdots, E^n))$, each E^i enumerating $e^i_0, e^i_1, \cdots e^i_{k_i}$. Let $k = min(k_1, \cdots, k_n)$.

$$\begin{array}{l} f(e^1_0, e^2_0, \cdots, e^{n-1}_0, e^n_0), \\ f(e^1_1, e^2_1, \cdots, e^{n-1}_1, e^n_1), \\ \cdots, \\ f(e^1_k, e^2_k, \cdots, e^{n-1}_k, e^n_k) \end{array}$$

Injective enumerators

$$\begin{aligned} \widehat{E} : \mathbb{N} &\to D \\ n &\mapsto e_n \end{aligned}$$

A sequence $\widehat{E}$ is an application from $\mathbb{N}$ to some domain D.
Not necessarily injective : we may have $e_i = e_j$ with $i \neq j$.
Example : $e_0 = 1, e_1 = -1, e_2 = 1, e_3 = -1, \ldots$.
For simplifying the proofs, it is easier to assume that enumerators are injective.
This yields no loss of generality :
a non injective enumerator may always be transformed into an injective one by putting it in parallel with an enumerator of $\mathbb{N}$.

Exemple of the parallel enumerator

```
ENUM> (setq *naturals* (make-inductive-enumerator 0 #'1+))
#<INDUCTIVE-ENUMERATOR {100C836033}>
ENUM> (collect-n-enum *naturals* 10) => (0 1 2 3 4 5 6 7 8 9)
ENUM> (setq *parallel*
            (make-parallel-enumerator (list *naturals* *abc*)))
#<PARALLEL-ENUMERATOR {10020EE5C3}>
ENUM> (collect-enum *parallel*) => ((0 A) (1 B) (2 C))
ENUM> (setq *parallel*
            (make-parallel-enumerator
              (list *naturals* (make-constant-enumerator 'a))))
#<PARALLEL-ENUMERATOR {10020F0893}>
ENUM> (collect-n-enum *parallel* 10)
((0 A) (1 A) (2 A) (3 A) (4 A) (5 A) (6 A) (7 A) (8 A) (9 A))
```

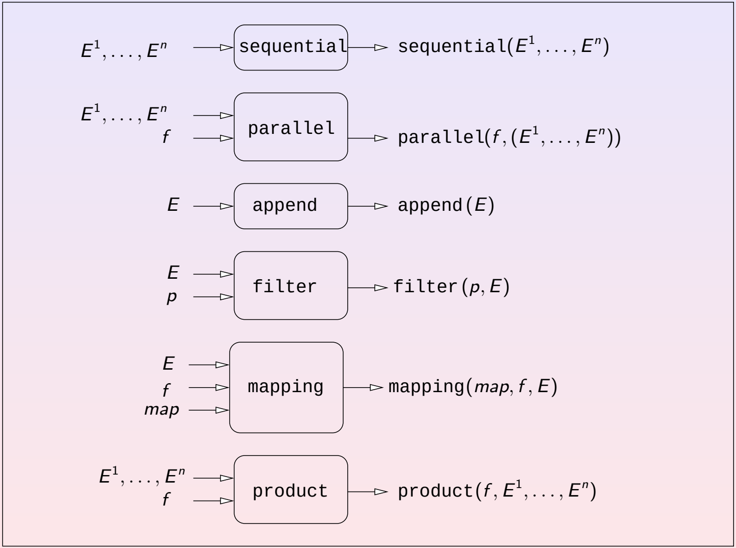
Application : implementation of the map function

```
(defun map (result-type fun &rest sequences)
  (let ((enumerator
          (make-parallel-enumerator
           (mapcar
            (lambda (s)
              (make-sequence-enumerator s))
            sequences)
           :fun (lambda (tuple)
                  (apply fun tuple)))))
    (if (null result-type)
        nil
        (coerce (collect-enum enumerator) result-type))))

ENUM> (map 'list #'list '(1 2 3) #(a b c d))
((1 A) (2 B) (3 C))
```

Application : Erathostenes's sieve

```
(defclass erathostenes (unary-relying-enumerator)
  ((enum :type abstract-enumerator :accessor enum
         :initform (make-inductive-enumerator 2 #'1+))))

(defmethod next-element ((e erathostenes))
  (let ((prime (next-element (enum e))))
    (setf (enum e)
          (make-instance
           'filter-enumerator
           :enum (enum e) :fun (lambda (n) (plusp (mod n prime)))))
    prime))

(defun make-erathostenes-enumerator () (make-instance 'erathostenes))

ENUM> (setq *e* (make-erathostenes-enumerator))
#<ERATHOSTENES {100C268E03}>
ENUM> (collect-n-enum *e* 10)
(2 3 5 7 11 13 17 19 23 29)
ENUM> (collect-n-enum *e* 10 :init nil)
(31 37 41 43 47 53 59 61 67 71)
```

Relying enumerators



Product enumerators

Let $E^1, \ldots, E^p$ be nonempty injective enumerators s.t each E^i enumerates

- $e^i_0, e^i_1, \ldots$ if E^i is infinite
- $e^i_0, e^i_1, \ldots, e^i_{c^i-1}$ where $c^i = card(E^i)$ otherwise.

Let $T^p = \widehat{E^1} \times \widehat{E^2} \ldots \times \widehat{E^p}$ be the cartesian product of the the $\widehat{E^i}$s.

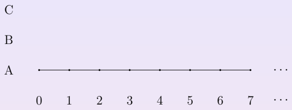
$T^p = \{(e^1_{j^1}, e^2_{j^2}, \ldots, e^p_{j^p}) \mid \forall i \in [1, p], e^i_{j^i} \in E^i\}$.
E^is injective $\Rightarrow$ tuples $(e^1_{j^1}, e^2_{j^2}, \ldots, e^p_{j^p})$ with distinct indices are distinct.

If every E^i is finite, $card(T^p) = \Pi^p_{i=1} c^i$ otherwise T^p is infinite.

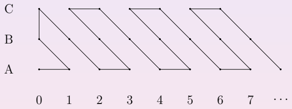
Multiple ways of ordering T^p so multiple possible enumerators of T^p.

Fairness property

Necessity of a fair ordering as soon as one of the E^i is infinite.



An unfair ordering `*naturals*` $\times$ `*abc*`



A fair diagonal ordering of `*naturals*` $\times$ `*abc*`

```
ENUM> (collect-n-enum
         (make-diagonal-product-enumerator *naturals* *abc*)
         20)
((0 A) (1 A) (0 B) (0 C) (1 B) (2 A) (3 A) (2 B) (1 C) (2 C)
 (3 B) (4 A) (5 A) (4 B) (3 C) (4 C) (5 B) (6 A) (7 A) (6 B))
```

This diagonal ordering was already there in 2012.

Bidirectional enumerators

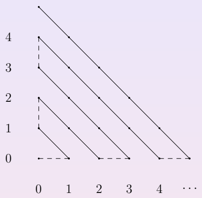
A bidirectional enumerator can move forward and backward.
It has a way and operations to deal with it.

```
(defun next-element-p (B) (way-next-element-p (way B) B))
(defun next-element (B) (way-next-element (way B) B))
```

```
ENUM> (setq *b-naturals* (make-bidirectional-enumerator *naturals*))
#<BIDIRECTIONAL-ENUMERATOR {100D46E113}>
ENUM> (way *b-naturals*) => 1
ENUM> (next-element *b-naturals*) => 0
ENUM> (next-element *b-naturals*) => 1
ENUM> (next-element *b-naturals*) => 2
ENUM> (next-element *b-naturals*) => 3
ENUM> (latest-element *b-naturals*) => 3
ENUM> (way-next-element -1 *b-naturals*) => 2
ENUM> (next-element *b-naturals*) => 3
ENUM> (invert-way *b-naturals*) => -1
ENUM> (next-element *b-naturals*) => 2
ENUM> (next-element *b-naturals*) => 1
ENUM> (way-next-element 1 *b-naturals*) => 2
ENUM> (next-element *b-naturals*) => 1
```

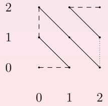
Diagonal ordering of binary product : sliding and corner steps

A pair of consecutive elements of an enumerator E is called a step.



A diagonal ordering of $\mathbb{N} \times \mathbb{N}$

Dashed line : sliding step
With finite enumerators also corner steps (dotted line).



A diagonal ordering of $[0, 2] \times [0, 2]$

Distance between tuples

In an ordering of the cartesian product T^p, we define a distance between two enumerated tuples $t_j = (e^1_{j^1}, \ldots, e^p_{j^p})$ and $t_k = (e^1_{k^1}, \ldots, e^p_{k^p})$ as $d(t_j, t_k) = \Sigma_{i=1}^{p} \mid k^i - j^i \mid$.

To simplify, from now on we assume that every E^i enumerates integers starting from 0.

A list of indices $(j^1, \cdots, j^p)$ will be the same as the tuple $(e^1_{j^1}, \ldots, e^p_{j^p}) : \forall i \in [1;p], \forall j, e^i_j = j$

$d(\texttt{(0 0)}, \texttt{(1 0)}) = 1, d(\texttt{(0 2)}, \texttt{(1 1)}) = 2$

Each enumerator is like the wheel of a mechanical counter which can be moved forward or backward of one or more times.
distance between two tuples = sum of sizes of moves of the wheels

d-orderings of a cartesian product T^p

size of a step (t_j, t_{j+1}) : $d(t_j, t_{j+1})$.
An ordering of T^p is a d-ordering if the size of each step is atmost d.
Our aim is to have a 2-ordering of T^p, $\forall p \geq 2$.

For $p = 2$, a diagonal ordering is a 2-ordering because each enumerator moves at most one notch forward or backward.

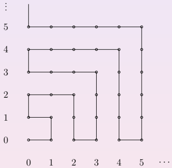
1-orderings for $p = 2$

There exist a 1-ordering for all binary products.

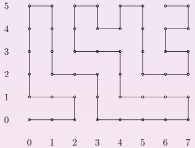
not feasible in our setting for arbitrary (finite or non finite) enumerators

need to know one step in advance whether we have reached the end of a finite enumerator to turn around before it is too late.

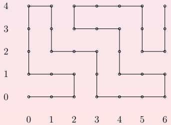
`next-element-p` not enough



$\mathbb{N} \times \mathbb{N}$



$[0, 7] \times [0, 5]$



$[0, 6] \times [0, 4]$

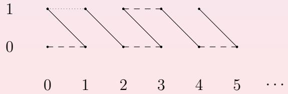
depends on the parity of the size of the finite enumerators

2-orderings

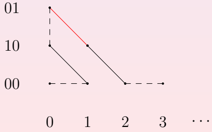
Our aim is to have a 2-ordering of T^p.
Recursive use of the binary diagonal $\Rightarrow$ p-ordering !

```
ENUM> (defparameter *e2* (make-list-enumerator '(0 1))) => *E2*
ENUM> (defparameter *p2*
        (make-diagonal-product-enumerator *naturals* *e2*)) => *P2*
ENUM> (collect-n-enum *p2* 11)
((0 0) (1 0) (0 1) (1 1) (2 0) (3 0) (2 1) (3 1) (4 0) (5 0) (4 1))
ENUM> (defparameter
          *p3*
        (make-diagonal-product-enumerator *e2* *p2* :fun #'cons))
*P3*
ENUM> (collect-n-enum *p3* 5)
((0 0 0) (1 0 0) (0 1 0) (0 0 1) (1 1 0))
```

$d($`(0 0 1)`$,$ `(1 1 0)`$) = 3$



$\mathbb{N} \times [0, 1]$



$\mathbb{N} \times [0, 1] \times [0, 1]$

leveled orderings of T^p

level : subset of tuples with identical height.
height of a tuple $t = (e^1_{j^1}, e^2_{j^2}, \ldots, e^p_{j^p}) \in T^p$ is the sum of the indices of the elements in the $\widehat{E^i}$: $h(t) = \Sigma_{i=1}^{p} j^i$
Note that with our hypothesis (each $\widehat{E^i}$ is a prefix of $\mathbb{N}$), the height of a tuple is the sum of its elements.

h^{th}-level of T^p : set of tuples with height h (denoted by L_h).

An ordering of T^p is leveled if it traverses the levels $L_0, L_1, \ldots$ in the increasing order of levels $L_0, L_1, \ldots$ (without constraint so far on the order of enumeration inside a level).

Leveled enumerators

leveled enumerators follow a leveled ordering : $L_0, L_1, \cdots$

A step giving a change of level is called a major step.

A step inside a level is called a minor step.

leveled enumerators have predicate :

- `minor-step-p (E)`

which returns `T` if the next step (`next-element`) does not change the level (`NIL` otherwise).

In other words, it returns `false` when we are done with the enumeration of the current level.

Bidirectional leveled enumerators

bidirectional leveled enumerator : leveled and bidirectional

forward : levels enumerated in increasing order : $L_0, L_1, \cdots$

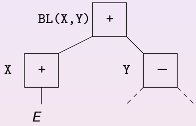
backward : levels enumerated in decreasing order : $L_i, L_{i-1}, \cdots$

```
ENUM> (defparameter *e3* (make-list-enumerator '(0 1 2)) => *e3*
ENUM> (defparameter
        *e* (make-bidirectional-leveled-enumerator *e3* *e3*)) => *E*
ENUM> (next-element *e*) => (0 0) ; level 0
ENUM> (next-element *e*) => (1 0) ; level 1
ENUM> (next-element *e*) => (0 1)
ENUM> (next-element *e*) => (0 2) ; level 2
ENUM> (next-element *e*) => (1 1)
ENUM> (next-element *e*) => (2 0)
ENUM> (next-element *e*) => (2 1) ; level 3
ENUM> (next-element *e*) => (1 2)
ENUM> (next-element *e*) => (2 2) ; level 4
ENUM> (invert-way *e*) => -1
ENUM> (next-element *e*) => (2 1) ; level 3
ENUM> (next-element *e*) => (1 2)
ENUM> (next-element *e*) => (0 2) ; level 2
```

Product of bidirectional and bidirectional leveled

Let X be a bidirectional enumerator and Y be a bidirectional leveled enumerator,

We define D = BL(X, Y), the leveled bidirectional product of X and Y.



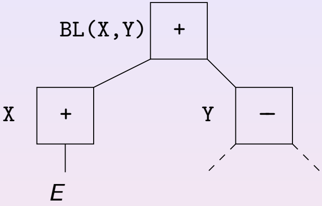
When D = BL(X, Y) is created, the initial way of X is set to +1 and the initial way of Y is set to −1.
The minor steps (same level) are diagonal steps
The major steps are either sliding or corner steps.
depending on the way of the enumerator the level increases or decreases by 1.

Code for the product

```lisp
(defun minor-step-p (D)
;; precondition (next-element-p D)
  (and (next-element-p (enum-y D))
       (or (next-element-p (enum-x D))
           (minor-step-p (enum-y D)))))

(defun way-next-element-p (way D)
  (or (way-next-element-p (way D) (enum-x D))
      (way-next-element-p (way(D) (enum-y D)))))

(defun sliding-step (X Y way)
;; precondition: X or Y can move in its way
  (if (next-element-p Y)
      (way-next-element way Y)
      (way-next-element way X))
  (invert-way X)
  (invert-way Y))

(defun corner-step (enum1 enum2 way)
  (when (plusp (* way (way enum1)))
    (psetf enum1 enum2 enum2 enum1))
  ;; enum1 is now the one that goes
  ;; in direction -way
  (invert-way enum1)
  ;; enum1 will move in direction way
  (next-element enum1)
  ;; enum2 will move in direction -way
  (invert-way enum2))
```

```lisp
(defun way-next-element (way D)
  (let* ((enum-x (enum-x enum))
         (enum-y (enum-y enum))
         (next-x (next-element-p enum-x))
         (next-y (next-element-p enum-y)))
   (cond
     ((and next-y (minor-step-p enum-y))
     ;; lower-level minor step
      (next-element enum-y))
     ((and next-y next-x) ; minor-step on level
     ;; each one makes a major in its way
      (next-element enum-x) (next-element enum-y))
     ;; major step
     ((not (or next-x next-y))
      (corner-step enum-x enum-y way))
     (t (sliding-step enum-x enum-y way))))
  (latest-element enum))

(defun latest-element (D)
  (cons (latest-element(enum-x D)
    (latest-element (enum-y D)))))
```

Properties of the bidirectional leveled product



`BL(X, Y)` is a bidirectional leveled enumerator.

bidirectional : by construction

leveled : by proof (see the paper)

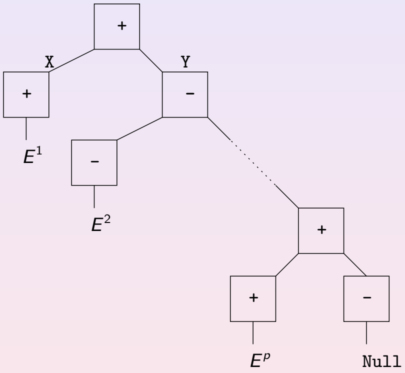
Moreover :

Lemma : if `Y` is a 2-ordering then so is `BL(X, Y)`

Proof : see the paper

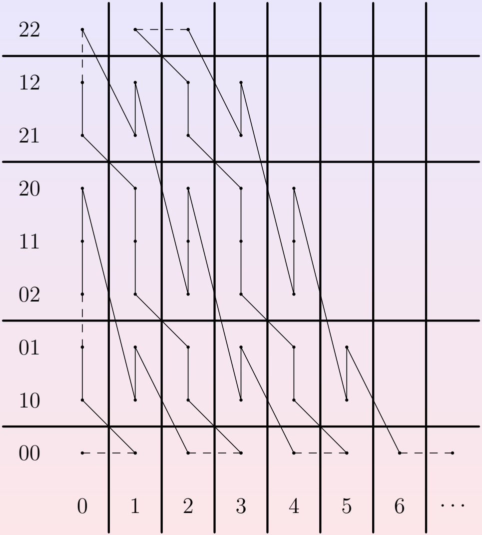
Bidirectional leveled enumerator for T^p

$\mathtt{BL}(E^1, \mathtt{BL}(E^2, \mathtt{BL}(\ldots, \mathtt{BL}(E^p, \mathtt{Null}))))$



Proof by induction that it is a leveled enumerator and a 2-ordering.

Example of a bidirectional leveled product



The leveled 2-ordering of $\mathbb{N} \times [0,2] \times [0,2]$

Conclusion and Perspectives

- Recursive use of bidirectional leveled product gives a fair and 2-ordering of the cartesian product T^p.
- Need for a non-recursive version to avoid stack exhaustion
- `Enum` package (2300 lines)
- Self-contained but not yet available separately

Thank you for your attention !