From lexical trees to effective Eilenberg machines: the Zen toolkit for computational linguistics

Gérard Huet

Emeritus, Inria Paris-Rocquencourt

Linguistics Department,
Stanford University
January 16th, 2014

What is Zen?

- Zen was designed 10 years ago as a CL toolkit
- In the form of a library in the functional language ML
- Mostly simple data structures and algorithms
- "Do it yourself" kind of toolkit
- Open source free software as LGPL Ocaml library

Data structures

Data structures are mathematical structures usable as structural data representations fit for efficient processing by effective algorithms.
The first data structure is *product*, used to construct pairs of values.

```
# (1,True);
- : (int * bool) = (1, True)
```

Pairing, noted infix as `(_,_)`, is doubly polymorphic:

```
# fun x y -> (x,y);
- : 'a -> 'b -> ('a * 'b) = <fun>
```

ML is λ-calculus typed in the polymorphic extension of Church's simple theory of types, with a *let* operator, conditional and recursion. A primitive integer library provides efficient arithmetic. It is Turing-complete, and permits principal typing inference.

Recursive type definitions

Enumeration types just list their constructors.

```
# type boolean = [ True | False ];
```

Recursive type definitions

Enumeration types just list their constructors.

```
# type boolean = [ True | False ];
```

Lists have two constructions, Nil and Cons, with specific syntax.

```
# [];
- : list 'a = []
```

Recursive type definitions

Enumeration types just list their constructors.

```
# type boolean = [ True | False ];
```

Lists have two constructions, Nil and Cons, with specific syntax.

```
# [];
- : list 'a = []

# [ 1 :: [] ];
- : list int = [1]
```

Recursive type definitions

Enumeration types just list their constructors.

```
# type boolean = [ True | False ];
```

Lists have two constructions, Nil and Cons, with specific syntax.

```
# [];
- : list 'a = []

# [ 1 :: [] ];
- : list int = [1]

# [ 1; 3; 7 ];
- : list int = [1; 3; 7]
```

Recursive type definitions

Enumeration types just list their constructors.

```
# type boolean = [ True | False ];
```

Lists have two constructions, Nil and Cons, with specific syntax.

```
# [];
- : list 'a = []

# [ 1 :: [] ];
- : list int = [1]

# [ 1; 3; 7 ];
- : list int = [1; 3; 7]

value rec nth n = fun (* n>0 *)
  [ [] -> failwith "index out of scope"
  | [ x :: l ] -> if n=1 then x else nth (n-1) l
  ];
```

Searching with a key: a-lists

```
# type alist 'a 'b = list ('a * 'b);
type alist 'a 'b = list ('a * 'b)
```

Trees and forests

```
type tree = [ Tree of forest ]
and forest = list tree
;
```

Trees and forests

```
type tree = [ Tree of forest ]
and forest = list tree
;

value tree1 =
  Tree [ Tree [ Tree [ Tree []
                     ; Tree [ Tree []
                            ; Tree []
                            ]
                     ]
               ; Tree [ Tree []
                      ; Tree []
                      ]
               ]
       ; Tree []
       ]
;
```

Trees and forests

We shall index in a tree with a *path*, encoded as a list of positive integers.

```
type path = list int;
value rec access t path = match t with
  [ Tree f -> match path with
      [ [] -> t
      | [ n :: rest ] -> access (nth n f) rest
      ]
  ];
```

Trees and forests

We shall index in a tree with a *path*, encoded as a list of positive integers.

```
type path = list int;
value rec access t path = match t with
  [ Tree f -> match path with
      [ [] -> t
      | [ n :: rest ] -> access (nth n f) rest
      ]
  ];

access tree1 [ 1; 1 ];
access tree1 [ 1; 1; 2 ];
access tree1 [ 3 ];
```

Our trees 'store' finite sets of sequences of natural numbers, closed under two operations, left and up, where left means 'left sibling', and up means 'ancestor'. Such sets are shared by initial prefixes of the sequences.

The contents of a tree

The following tree traversal is an exercise in recursive list-processing, using the primitives of the Ocaml List library.

```
value contents = traverse []
  where rec traverse pref = fun
  [ Tree l -> let (down,_) =
    let f (l,n) t = (l @ (traverse [n :: pref] t),n+1) in
    List.fold_left f ([],0) l in [ List.rev pref :: down ]
  ];
```

```
contents tree1;

[[];
    [1];
       [1; 1];
          [1; 1; 1];
          [1; 1; 2];
             [1; 1; 2; 1];
             [1; 1; 2; 2];
       [1; 2];
          [1; 2; 1];
          [1; 2; 2];
    [2]
]
```

Note that the result of `contents` lists the sequences in lexicographic ordering.

Data, information, data structure

We usually think of a data structure as some container shape, holding data at certain places. Here, it looks like the tree data structure is not holding any data, it is just a container shape. But this shape itself is some kind of intrinsic information, and this is what the `contents` function reveals. This implicit data is simply the *shape* of the structure. Reading out the data is just navigating successfully in the structure.

Data, information, data structure

We usually think of a data structure as some container shape, holding data at certain places. Here, it looks like the tree data structure is not holding any data, it is just a container shape. But this shape itself is some kind of intrinsic information, and this is what the `contents` function reveals. This implicit data is simply the *shape* of the structure. Reading out the data is just navigating successfully in the structure. The shape is the data !

Data, information, data structure

We usually think of a data structure as some container shape, holding data at certain places. Here, it looks like the tree data structure is not holding any data, it is just a container shape. But this shape itself is some kind of intrinsic information, and this is what the `contents` function reveals. This implicit data is simply the *shape* of the structure. Reading out the data is just navigating successfully in the structure. The shape is the data ! This is reminiscent of Marshall McLuhan's maxim: The medium is the message !

Sparse trees

We now make our structure sparse, by removing the closure conditions. This is achieved with two devices. Firstly, every node of the tree is marked with a Boolean value, indicating whether the path that indexes it is in the set or not. Secondly, siblings of a given node are labeled with an explicit integer key, instead of being implicitly labeled according to their position. The labels of common siblings are assumed to be in increasing order, but are not necessarily a sequence of contiguous natural numbers. Such sparse structures are called *tries*.

```
type trie = [ Trie of (bool * arcs) ]
and arcs = list (int * trie)
;
```

You may think of a trie as a tree-like graph, with integers labeling the arcs.

Example

For instance, the set of integer sequences

```
[[1]; [3; 1; 18]; [3; 1; 20]]
```

is canonically represented by the trie:

```
value trie1 =
Trie(False,
     [(1,Trie(True,[]))
     ;(3,Trie(False,[(1,Trie(False,[(18,Trie(True,[]))
                                    ;(20,Trie(True,[]))
                                    ]))
                     ]))
     ]);
```

Contents of a trie

Similar algorithm to the one giving the contents of a tree:

```
value contents = traverse []
  where rec traverse pref = fun
  [ Trie (b,l) -> let down =
    let f l (n,t) = l @ (traverse [ n :: pref ] t) in
    List.fold_left f [] l in
       if b then [ List.rev pref :: down ] else down
  ]
;
```

and we may check:

```
contents trie1;
(* [[1]; [3; 1; 18]; [3; 1; 20]] *)
```

Words

```
type letter = int (* letter or phoneme *)
and word = list letter
;
```

Note that we are not using for our word representations the ML type of strings (which in OCaml are arrays of characters/bytes). Strings are convenient for English texts (using the 7-bit low half of ASCII) or other European languages (using the ISO-LATIN subsets of full 8-bit ASCII), and they are more compact than lists of integers, but basic operations like pattern matching are awkward, and they limit the size of the alphabet to 256, which is insufficient for the treatment of many languages' written representations, specially for multi-lingual documents.

Words (continued)

New format standards such as Unicode have complex primitives for their manipulation, and are better reserved for interface modules than for central morphological operations. We could have used an abstract type of characters, leaving to module instantiation their precise definition, but here we chose the simple solution of using machine integers for their representation, which is sufficient for large alphabets (in Ocaml, this limits the alphabet size to 1073741823), and to use conversion functions [encode] and [decode] between words and strings. In the Sanskrit application, we use the first 50 natural numbers as the character codes of the Sanskrit phonemes, whereas string translations take care of transliterations, roman diacritics notations such as IAST, and encodings of *devanāgarī* syllabic glyphs.

Encode/decode

Here is Zen's Ascii module:

```
(* encode : string -> word *)
(* decode : word -> string *)
value encode str = List.map int_of_char (List2.explode str)
and decode word = List2.implode (List.map char_of_int word)
;
```

Let us consider a slight variant of decode:

```
value my_decode w = List2.implode
   (List.map (fun n -> char_of_int (n+96)) w) ^ "\n";
let display w = print_string (my_decode w) in
List.iter display (contents trie1);
```

Encode/decode

Here is Zen's Ascii module:

```
(* encode : string -> word *)
(* decode : word -> string *)
value encode str = List.map int_of_char (List2.explode str)
and decode word = List2.implode (List.map char_of_int word)
;
```

Let us consider a slight variant of decode:

```
value my_decode w = List2.implode
   (List.map (fun n -> char_of_int (n+96)) w) ^ "\n";
let display w = print_string (my_decode w) in
List.iter display (contents trie1);
```

```
a
car
cat
```

Ah ah! `trie1` was hiding an English dictionary !

Dictionaries

We encode dictionaries as tries (also called *lexical trees*).
Looking up a word in the dictionary amounts to interpret the word as an access path in the trie, and to return the boolean at that location (or `False` if it is outside its tree domain).
You may also think of a trie as a deterministic finite-state automaton, whose states are the nodes (accepting states if they are labeled `True`).
Building dictionaries efficienty from lists of words is explained in the Zen notes with data structures giving fast access to focused trees called *zippers*.
Such dictionaries share the entries of the dictionary having a common prefix (such as 'car' and 'cat' in the example, since their common initial prefix 'ca' is shared). Furthermore, we can do better by sharing entries having a common suffix, using dynamic programming techniques.

Sharing

The Zen notes expose a universal sharing function, applying to **any** applicative data-structure. This generalizes the well-known technique that produces a directed acyclic graph (dag) from a tree, by bottom-up traversal with the help of a hashing function. Applied to tries, it yields the corresponding minimal automaton.
Applying the sharing function to a trie dictionary yields a compressed structure, while preserving its type as a `trie`, and thus preserving all the operations of the `trie` library.

Efficiency considerations

If we apply our technique to a list of 180000 English words, taken from an ASCII file of 2 MB, we obtain a trie of 4.5 MB, which shrinks to 1.1 MB by sharing. The whole operation takes about 1s on a 2MHz PC.
Measurements show that the time complexity is linear with the size of the lexicon This is consistent with algorithmic analysis, since it is known that tries compress dictionaries up to a linear entropy factor, and that perfect hashing compresses trees in dags in linear time.

Decorated tries

The next idea is to decorate our dictionaries with annotations, such as morphological analyses of the (inflected) words stored in them.

```
(* Tries storing decorated words. *)
type deco 'a = [ Deco of (list 'a * darcs 'a) ]
and darcs 'a = list (Word.letter * deco 'a)
;
```

Typically, the type variable `'a` will be instanciated by the structure of lemmas carrying the morphological features of the form coresponding to the access path to the node.
Decorated tries have the same operations as tries, of which they are a natural extension. In particular, they can be compressed with sharing of common morphological structures.

Differential words

The main problem is how to store succintly the stem that produces a given form. The idea is to denote the stem as a relative function of the inflected form, encoded as an *editing distance* in the structure.
To this end, we define a datatype `delta` of *differential words*. A *differential word* is a notation permitting to retrieve a word `w` from another word `w'` sharing a common prefix. It denotes the minimal path connecting the words in a trie, as a sequence of ups and downs: if $d = (n, u)$ we go up n times and then down along word u.

```
type delta = (int * word)  (* differential words *)
;
```

Diff

We compute the difference between w and w' as a differential word $diff\ w\ w' = (|w1|, w2)$ where $w = p.w1$ and $w' = p.w2$, with maximal common prefix p.

```
(* diff : word -> word -> delta *)
value rec diff w w' = match w with
  [ [] -> (0,w')
  | [ c :: r ] -> match w' with
      [ [] -> (length w,[])
      | [ c' :: r' ] -> if c = c' then diff r r'
                                  else (length w,w')
      ]
  ];
```

Patch (a tribute to UNIX)

Conversely, w' is retrieved from w and $d = diff\ w\ w'$ as $patch\ d\ w$:

```
(* patch : delta -> word -> word *)
value patch (n,dw) w =
  let p = List2.truncate n (mirror w) in
  List2.unstack p dw;
```

Note how `diff` and `patch` are related to their UNIX homonyms.

Example: English plural

```
cats : { plural cat }
dogs : { plural dog }
```

Both share the morphology `{ plural stem }` with `stem` represented as the differential word `(1,[])`. Along with all regular plurals. When sharing the dictionary of forms, all the lemmas of regular plurals are shared. Thus, the structure holding the morphological information is of same size as the dictionary of stems. We just need a few extra bits for irregular pairs (mouse,mice), (spy, spies), (leaf, leaves), (foot, feet), (ox, oxen), (sheep,sheep), etc.

Derivational morphology

The same ideas may be applied to derivational morphology, where stems are represented relatively to roots. According to morphological processes, the ideas of differential words may be adapted for succinct encoding.
Note that lemmatization/stemming is just look-up in the structure of decorated dictionaries of inflected forms.

More of the same

Our decorated tries give a systematic treatment to finite-state morpho-phonetics. They can be adapted to more complex operations expressible as regular relations, such as segmentation and shallow parsing.
This technology has given rise to a new paradigm of *relational programming* with effective Eilenberg machines. The *sandhi viccheda* algorithm of the Sanskrit Heritage platform is a prime example of this methodology.

The 3 golden rules of non-deterministic programming

The 3 golden rules of non-deterministic programming

- Identify well your search state space

The 3 golden rules of non-deterministic programming

- Identify well your search state space
- Prove termination or at least fairness

The 3 golden rules of non-deterministic programming

- Identify well your search state space
- Prove termination or at least fairness
- Represent states as non-mutable data

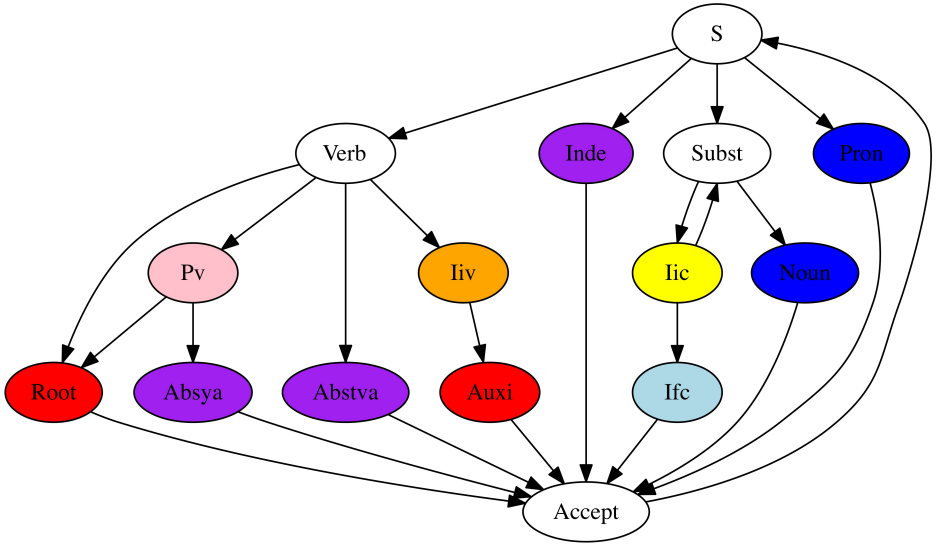
The reactive engine (terminating case)

Let us assume that we search for all elements y such that xR^*y for a given x, where R is a relation which is locally finite and noetherian.
It is easy to enumerate all the solutions using a iterative bottom-up search procedure in a coroutine fashion.
This is extendable to solving problems defined in terms of *rational relations* obeying certain convergence conditions. The Zen module *Reactive engine* is adapted to this task, and a completeness of its simulation may be formally proved in the proof assistant Coq.
This offers an elegant, efficient and complete solution to the Segmentation of classical Sanskrit, implementing *sandhi viccheda* in the Sanskrit Heritage engine.

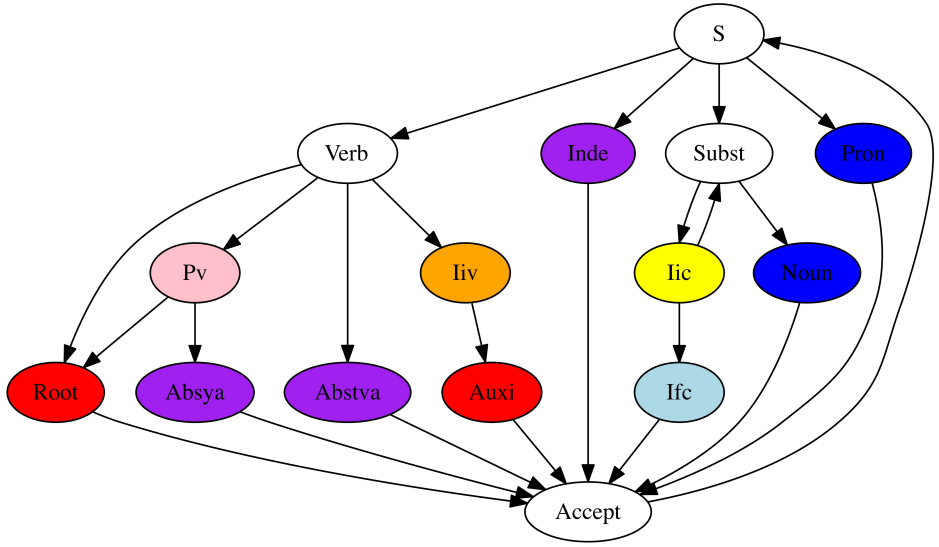
Eilenberg Machines

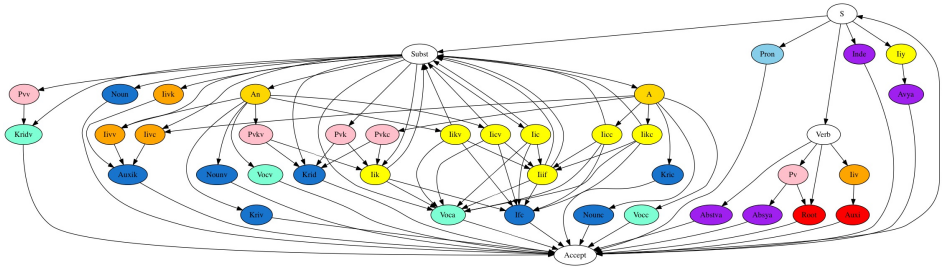
These results may be set in the general theory of *Eilenberg Machines*.
First, Benoît Razet coined the notion of *Finite Eilenberg Machine*, and showed that the reactive engine could be generalized to their deterministic simulation. He went on to define in his Ph.D. general *Effective Eilenberg Machines*, using streams to represent progressive relations, and generalizing the reactive engine with selection tactics. These machines are modular, and may be composed both horizontally and vertically.

Sanskrit morphology as a 2-level machine



Sanskrit morphology as a 2-level machine





Eilenberg Machines for Relational Programming

Actions of the machine are effective relations, semantically attached to the action generators. The Control component is the transition graph of a non-determinitic automaton on this alphabet, properly represented as a decorated trie with virtual addresses for representing loops. The Data component computes partial recursive relations on a computation domain where sets of values are represented as lazy streams. Pratt's Action algebras offer a proper basis for the semantics of both Control and Data.

This paradigm of relational programming offers an elegant mixture of finite-state methods and Turing-complete computations. It may be put to use for general non-deterministic computations, as we saw, but may be envisioned as a general paradigm for distributed programming as well.

Icalp 1972

Benoît Razet