Computational Linguistics

From Zen to Aum

Gérard Huet

Chalmers University, May 21st, 2004

The Zen toolkit - Generic technology

A few specific applicative techniques:

- Local processing of focused data
- Sharing
- Lexical trees
- Differential words
- Finite transducers as lexicon morphisms
- Search by resumption coroutines
- Multiset ordering convergence

Automata Mista - AuM

We represent finite-state automata by a mixed structure - a deterministic skeleton decorated by non-deterministic transitions.

The first component is a forest of lexical trees, used as covering trees of the state transitions graph. The rest of the transitions is represented as annotations stating that on a certain input (a word possibly empty, allowing ϵ-transitions), the automaton goes to a state designated by a virtual address. There are two kinds of addresses, local and global. A global address is given by an integer (indexing into the forest array) and a word. A local address has the same structure, but now acts as a differential word. Its first component indexes into an array representing the access path in the current tree (necessary because of sharing).

Differential words

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

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 tree, as a sequence of ups and downs: if $d = (n, u)$ we go up n times and then down along word u.

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.

The converse of `diff : word -> word -> delta` is `patch : delta -> word -> word`: w' may be retrieved from w and $d = diff\ w\ w'$ as $w' = patch\ d\ w$.

The automaton structure

```
type input = word;

type delta = (int * word)
and address = [ Global of delta | Local of delta ];

type auto =  [ State of (bool * deter * choices) ]
and deter = list (letter * auto)
and choices = list (input * address);

type automaton = (array auto * delta);

type backtrack = (input * delta * choices)
and resumption = list backtrack; (* coroutine resumptions *)
```

Completeness

Every non-deterministic automaton (possibly with ϵ trasitions) may be represented as a flat aum (with empty deterministic structure).

Every deterministic automaton may be represented as an aum whose choice annotations `State(b,[],[([],address)])` do not give rise to backtrack.

Every aum has a minimal representation, obtained by maximal sharing. N.B. Sharing the local virtual adresses does not necessarily correspond by equivalence by bisimulation.

The transducer structure

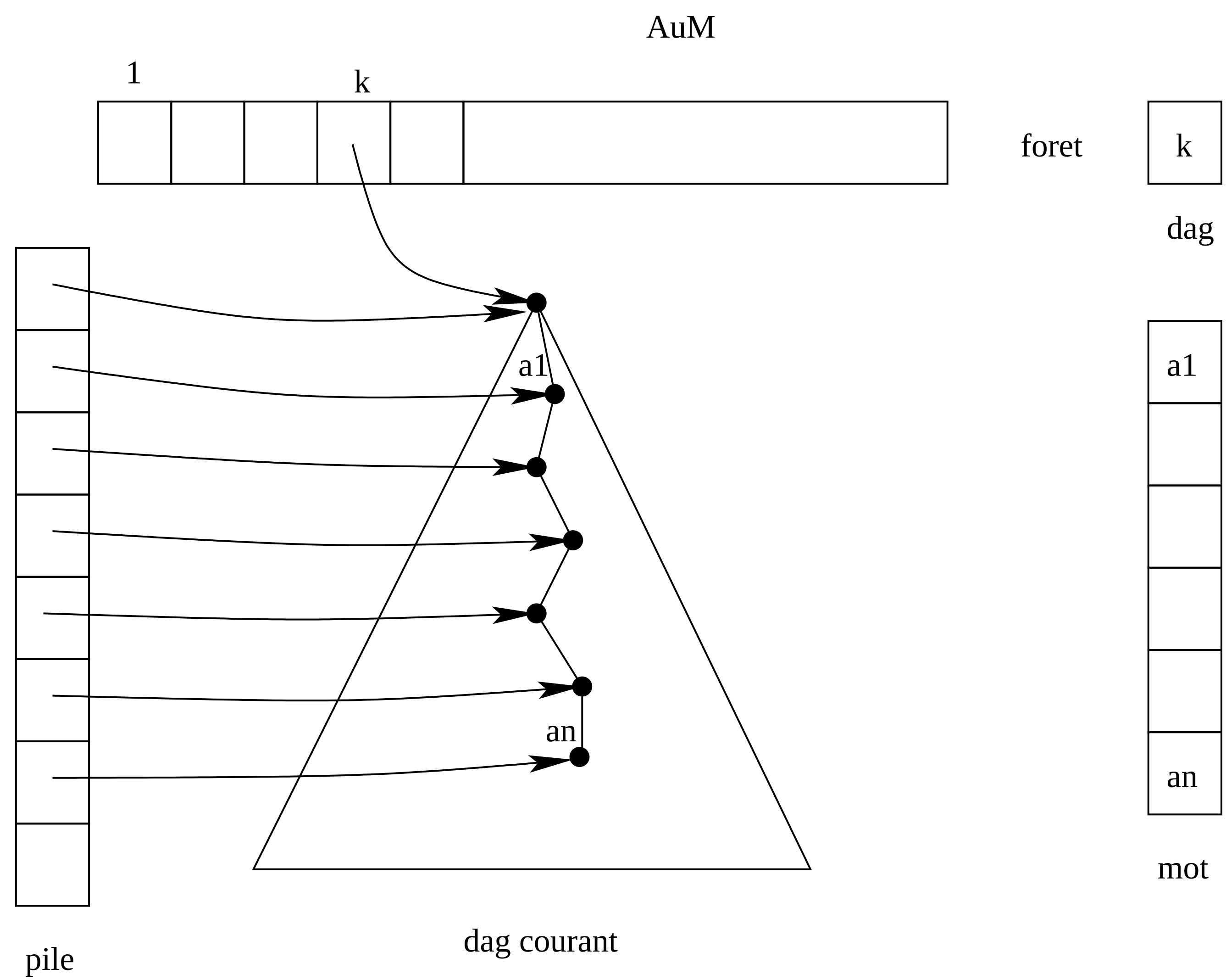
```
type input = word and output = word;

type delta = (int * word)
and address = [ Global of delta | Local of delta ];

type trans =  [ State of (bool * deter * choices) ]
and deter = list (letter * trans)
and choices = list (input * output * address);

type transducer = (array trans * delta);

type backtrack = (input * output * delta * choices)
and resumption = list backtrack; (* coroutine resumptions *)
```


AuM
1
k
foret
k
dag
a1
an
a1
an
mot
pile
dag courant

Memorisation of the current access

The access stack $[s_n; s_{n-1}; ...s_0]$ is necessary, to interpret local virtual addresses. It may be convenient to store as well the current access *word* $word = [a_n; ...a_1]$, stacked and unstacked along the local accesses. We may thus distinguish two output constructors: `Absolute of word` et `Relative of delta`. In the last case, output is computed by *patch* applied to *word*.

Applications:

- Inflected forms dictionary used as lemmatizer (regular plural: $(\delta = (1, ['s']))$
- Unglue $(\delta = (0, []))$
- Segment $(\delta = (0, u))$

Modular aums

An aum is given by a pair in `(array auto * delta)`.

We make them modular by making the global addresses relocatable, and possibly interpreting success states by continuations. Continuations are implemented as ϵ-transitions, i.e. extra `choices`, with empty input.

Now it is easy to compile regular expressions into aums, as follows:

- The base case is any aum, its size the size of its array
- if $A = (array_A, delta_A)$ is of size a and $B = (array_B, delta_B)$ is of size b, $A \cdot B$ is obtained by relocating B by a, continuing A by $a + delta_B$, starting at $delta_A$, of size $a + b$.
- if $A = (array_A, delta_A)$ is of size a and $B = (array_B, delta_B)$ is of size b, $A + B$ is obtained by relocating B by a, starting at $a + b + 1$, where we put

$State(False, [], [([], delta_A); ([], a + delta_B)])$, of size $a + b + 1$.

- if $A = (array_A, delta_A)$ is of size a, then $A*$ is obtained by continuing A by $delta_A$, making its starting node accepting, of size a.

These transformations ought to be effected *before* sharing.

Conclusion

Automata mista offer an elegant applicative solution to many finite-state processing problems, typically the treatment of lexicon representation, phonology, morphology and segmentation in computational linguistics. The deterministic spanning tree of their state space is then naturally the dictionary of inflected forms of words, which is thus placed at the center of the computer treatment of language.