JFP **15** (4): 573–614, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796804005416 Printed in the United Kingdom

A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger

GÉRARD HUET
INRIA Rocquencourt, BP 105, F-78153 Le Chesnay Cedex
(*e-mail:* `Gerard.Huet@inria.fr`)

Dedicated to Rod Burstall on the occasion of his 65th birthday

Abstract

We present the *Zen* toolkit for morphological and phonological processing of natural languages. This toolkit is presented in literate programming style, in the Pidgin ML subset of the Objective Caml functional programming language. This toolkit is based on a systematic representation of finite state automata and transducers as decorated lexical trees. All operations on the state space data structures use the *zipper* technology, and a uniform *sharing functor* permits systematic maximum sharing as dags. A particular case of *lexical maps* is specially convenient for building invertible morphological operations such as inflected forms dictionaries, using a notion of *differential word*. As a particular application, we describe a general method for tagging a natural language text given as a phoneme stream by analysing possible euphonic liaisons between words belonging to a lexicon of inflected forms. The method uses the toolkit methodology by constructing a non-deterministic transducer, implementing rational rewrite rules, by mechanical decoration of a trie representation of the lexicon index. The algorithm is linear in the size of the lexicon. A coroutine interpreter is given, and its correctness and completeness are formally proved. An application to the segmentation of Sanskrit by sandhi analysis is demonstrated.

Introduction

Understanding natural language with the help of computers, or computational linguistics, usually distinguishes a number of phases in the recognition of human speech. When the input is actual speech, the phonetic stream must be analyzed first as a stream of phonemes specific to the language at hand and then as a stream of words, taking into account euphony phenomena. Then this stream of words must be segmented into sentences, and tagged with grammatical features to account for morphological formation rules, then parsed into phrasal constituents, and finally analyzed for meaning through higher semantic processes such as anaphora resolution and discourse analysis. When the input is written text, it is often already segmented into words. The complexity and mutual interaction of the various phases vary widely across the variety of human languages.

The techniques used by computational linguistics involve statistical analysis methods, such as hidden Markov chains built by corpus data mining or by training, and logical analysis through formal language theory and computational logic. Despite the variety of approaches, two components are essential: a structured lexicon acting as a modular repository of grammatical information, and finite state automata and transducers tools. The Xerox rational relations library (Beesley & Karttunen, 2003) is typical of the latter.

Computational linguistics platforms fall usually in two categories. When they strive for robust algorithmic treatment of a realistic corpus, they are usually implemented in an imperative language such as C++. When they are more speculative experiments in linguistic modeling they are often implemented in some version of PROLOG. Although in the seventies LISP was a frequent implementation medium for computational linguistics, nowadays very few efforts use functional programming, a notable exception being Grammatical Framework (Ranta, 2003). We shall describe here the Zen toolkit for lexical, morphological and phonological processing, as the first layer in a generic computational linguistics platform in ML. We discuss the main design principles of this functional toolkit, and give in detail the algorithmic treatment of key structures. We present its application to a tough computational problem, the segmentation and morphological tagging of Sanskrit text.

The computational problems arising from the mechanical treatment of Sanskrit fall somewhere between speech recognition and the analysis of written text. The Sanskrit tradition is extremely attentive to the preservation of its correct utterance. Consequently, its written representation is actually a phonemic stream, where the words composing a sentence are glued together by euphonic transformation processes known as *sandhi* (juncture). A linguistic tradition going back more than 25 centuries was frozen by the grammarian Pāṇini in the 4th century B.C. In his treatise Aṣṭādhyāyī, grammatical rules for Sanskrit are laid out in great precision, in a formal rewriting notation that has been shown to be equivalent to context free grammars. There are actually two distinct *sandhi* operations. Firstly, internal *sandhi* is applied by morphological rules for word formation from stems, prefixes, and suffixes used in noun declension and verb conjugation. Secondly, external *sandhi* is applied to join words together in a sentence or stanza. Internal *sandhi* is a complex combination of phonetic rules, which may cascade with long-distance effects (such as retroflexion of nasals and sibilants). External *sandhi* is more regular and local. Neither operation is one-one, leading to complex non-determinism in their analysis. Such ambiguities are actually exploited by poets in *double entendre* statements.

The first analysis of a Sanskrit sentence consists thus in its segmentation into words by inversion of external *sandhi*, followed by the stemming or morphological analysis of these words into lexicon entries tagged with grammatical features. This last process is constrained by agreement within nominal chunks and by valency requirements (subcategorization) within verbal phrases. A further complication arises from compound nouns, which may agglutinate together an unbounded number of stems, applying at their juncture a phonetical process similar to external *sandhi*. The good news however is that the resulting stream of tagged lexical entries yields a first approximation to syntactic analysis, since word order is relatively unimportant for the meaning, thanks to the rich inflexional morphology.

We have built a prototype tagger for Sanskrit as an interpreter for a finite state machine mechanically produced from a structured lexicon by a regular transducer compiler. The lexicon contains inflected forms generated by internal *sandhi* from a stem dictionary annotated with grammatical information, whereas the regular transducer is specified with external *sandhi* rewrite rules. This methodology is described in a generic manner, independent from Sanskrit, using the Zen toolkit structures. We conclude with formal justification of the segmenting algorithm, independent of the non-determinism resolution strategy.

1 Pidgin ML

We shall describe our algorithms in a pidgin version of the functional language ML (Landin, 1966; Burstall, 1969; Gordon *et al.*, 1977; Paulson, 1991; Weis & Leroy, 1999). Readers familiar with ML may skip this section, which gives a crash overview of its syntax and semantics.

The core language has types, values, and exceptions. Thus, 1 is a value of predefined type int, whereas "FP" is a string. Pairs of values inhabit the corresponding product type. Thus: (1,"FP") : (int × string). Recursive type declarations create new types, whose values are inductively built from the associated constructors. Thus the Boolean type could be declared as a sum by: **type** bool = [True | False]; Parametric types give rise to polymorphism. Thus if x is of type t and l is of type (list t), we construct the list adding x to l as [x :: l]. The empty list is [], of (polymorphic) type (list α). Explicit type specification is rarely needed from the programmer, since principal types may be inferred mechanically.

The language is functional in the sense that functions are first class objects. Thus the doubling integer function may be written as **fun** x→x+x, and it has type int→int. It may be associated to the name double by declaring:
value double = **fun** x→x+x;
Equivalently we could write: **value** double x = x+x;
Its application to value *n* is written as '(double n)' or even 'double n' when there is no ambiguity. Application associates to the left, and thus 'f x y' stands for '((f x) y)'. Recursive functional values are declared with the keyword **rec**. Thus we may define factorial as:
value rec fact n = **if** n=0 **then** 1 **else** n*(fact (n−1));
Functions may be defined by pattern matching. Thus the first projection of pairs could be defined by:
value fst = **fun** [(x,y) →x];
or equivalently (since there is only one pattern in this case) by:
value fst (x,y) = x;
Pattern-matching is also usable in **match** expressions that generalize case analysis, such as: **match** l **with** [[] → True | _ → False], which tests whether list l is empty, using underscore as catch-all pattern.

Evaluation is strict, which means that x is evaluated before f in the evaluation of (f x). The **let** expressions allow the sharing of sub-computations. Thus '**let** x = fact 10 **in** x+x' will compute 'fact 10' first, and only once. An equivalent postfix **where** notation may be used as well. Thus the conditional expression '**if** b **then** e1 **else** e2' is equivalent to:
choose b **where** choose = **fun** [True → e1 | False → e2].

Exceptions are declared with the type of their parameters, like in:
exception Failure **of** string;
An exceptional value may be raised, like in: **raise** (Failure `"div␣0"`) and handled by a **try** switching on exception patterns, such as:
try expression **with** [Failure s →...].

Other imperative constructs may be used, such as references, mutable arrays, while loops and I/O commands, but we shall seldom need them. Sequences of instructions are evaluated in left to right regime in **do** expressions, such as:
do {e1; ... en}.

ML is a *modular* language (Burstall, 1984), in the sense that sequences of type, value and exception declarations may be packed in a structural unit called a **module**, amenable to separate treatment. Modules have types themselves, called *signatures*. Parametric modules are called *functors*. The algorithms presented in this paper will use in essential ways this modular structure, but the syntax ought to be self-evident. Finally, comments are enclosed within starred parens like *(* This is a comment *)*.

Readers uninterested in computational details may think of ML definitions as recursive equations over inductively defined algebras. Most of them are simple primitive recursive functionals. The more complex recursions of our automata coroutines will be shown to be well-founded by a combination of lexicographic and multiset orderings.

Pidgin ML definitions may actually be directly executed as Objective Caml programs (Leroy *et al.*, 2000), under the so-called revised syntax (de Rauglaudre, 2001). The following development may thus be used as the reference implementation of a core computational linguistics toolkit, dealing with lexical, phonological and morphological aspects. Readers interested in generic computational linguistics technology may skip Sanskrit technicalities, whereas readers interested primarily in Sanskrit tagging may skim over the Zen toolkit presentation. An extended version of this toolkit documentation is available as ESSLLI course notes (Huet, 2002).

2 Trie structures for lexicon indexing

Tries are tree structures that store finite sets of strings sharing initial prefixes. We assume that the alphabet of string representations is some initial segment of positive integers. Thus a string is encoded as a list of integers that will from now on be called a *word*. For our Sanskrit application, the Sanskrit alphabet comprises 50 letters, representing 50 phonemes. Finite state transducers convert back and forth lists of such integers into strings of transliterations in the roman alphabet, which encode themselves either letters with diacritics, or Unicode representations of the *devanāgarī* alphabet. Thus 1,2,3,4, etc. encode, respectively, a, ā, i, ī, etc.

2.1 Words and tries

type letter = int
and word = list letter;

We remark 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 words (using the 7-bit low half of ASCII) or other European languages (using the ISO-LATIN subsets), 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. New format standards such as Unicode have complex primitives for their manipulation (since characters have variable width in the usual encodings such as UTF-8), and are better reserved for interchange 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 shall use the first 50 natural numbers as the character codes of the Sanskrit phonemes, whereas string translations take care of roman diacritics notations, and encodings of *devanāgarī* characters and their ligatures. Thus the *anusvāra* nasalization dot turns, by contextual analysis, into either one of the five nasal consonants, or the so-called *original anusvāra* marking nasalization of the preceding vowel.

Tries (also called *lexical trees*) may be implemented in various ways. A node in a trie represents a string, which may or may not belong to the set of strings encoded in the trie, together with the set of tries of all suffixes of strings in the set having this string as a prefix. The forest of sibling tries at a given level may be stored as an array, or as a list if we assume a sparse representation. It could also use any of the more efficient representations of finite sets, such as search trees (Bentley & Sedgewick, 1997). Here we shall assume the simple sparse representation with lists[1], yielding the inductive type structure:

```
type trie = [ Trie of (bool × arcs) ]
and arcs = list (letter × trie);
```

The boolean value indicates whether the path to the current node belongs to the set of strings encoded in the trie. Note that letters decorate the *arcs* of the trie, not its *nodes*. For instance, the trie storing the set of words {[1; 2], [2], [2; 2], [2; 3]} standing for the strings {AB, B, BB, BC} is represented as

```
Trie(False ,[(1, Trie(False ,[(2, Trie(True ,[]))]));
             (2, Trie(True,  [(2, Trie(True ,[]));
                              (3, Trie(True  ,[]))]))])
```

This example exhibits one invariant of our representation, namely that the integers in successive sibling nodes are in increasing order. Thus a top-down left-to-right traversal of the trie lists its strings in lexicographic order. The algorithms below maintain this invariant.

[1] That is actually the original presentation of tries by René de la Briantais (1959).

2.2 Zippers

Let us show how to add words to a trie in a completely applicative way, using the notion of a zipper (Huet, 1997; 2003c). Zippers encode the context in which some substructure is embedded. They are used to implement applicatively destructive operations in mutable data structures. In the case of tries, we get:

```
type zipper =
  [ Top
  | Zip of (bool × arcs × letter × arcs × zipper)
  ]
and edit_state = (zipper × trie);
```

An edit_state *(z,t0)* stores the editing context as a zipper *z* and the current subtrie *t0*. We replace this subtrie by a trie *t* by closing the zipper with *zip_up z t* defined as:

```
(* zip_up : zipper → trie → trie *)
value rec zip_up z t = match z with
  [ Top → t
  | Zip(b,left,n,right,up) →
    zip_up up (Trie(b,unstack left [(n,t)::right]))
  ];
```

where the infix operations *::* and *@* are respectively the list constructor and append operator, and *unstack l r = (rev l)@ r* (sometimes called *rev_append*). We need two auxiliary routines. The first one, given an integer *n* and a list of arcs *l*, partitions *l* as *l1@l2* with *l1* maximum with labels less than *n*, in such a way that *zip n l = (rev l1,l2)*:

```
value zip n = zip_rec []
  where rec zip_rec acc l = match l with
  [ [] → (acc,[])
  | [((m,_) as current)::rest] →
      if m<n then zip_rec [current::acc] rest
      else (acc,l)
  ];
```

Its name stems from the fact that it looks for an element in an a-list while building an editing context in the spirit of a zipper, the role of *zip_up* being played by *unstack*. Notice the use of an *as* expression for naming a sub-pattern.

The second routine, given a word *c*, returns the singleton trie containing *c* as *trie_of c*:

```
value rec trie_of = fun
  [ [] → Trie(True,[])
  | [n::rest] → Trie(False,[(n, trie_of rest)])
  ];
```

2.3 Insertion and lookup

We are now ready to define the algorithm *enter : trie → word → trie*:

```
value enter trie = enter_edit (Top, trie)
  where rec enter_edit (z,t) =
    match t with [ Trie(b,l) → fun
      [ [] → if b then raise Redundancy
                else zip_up z (Trie(True,l))
      | [n::rest] → let (left, right) = zip n l
                      in match right with
          [ [] → zip_up (Zip(b,left,n,[],z)) (trie_of rest)
          | [(m,u)::r] →
            if m=n then enter_edit (Zip(b,left,n,r,z),u) rest
            else zip_up (Zip(b,left,n,right,z)) (trie_of rest)
          ]
      ]];
```

A variant could raise an exception *Redundancy* when attempting to enter a duplicate element.

Now, assuming the coercion *encode* from strings to words, we build a lexicon trie from a list of strings by function *make_lex*, using Ocaml's list library *fold_left* (the terminal recursive list iterator).

```
value make_lex =
  List.fold_left (fun lex c → enter lex (encode c)) empty
  where empty = Trie(False,[]);
```

Conversely, we get the set of words stored in a trie by function *contents*, which lists them in lexicographic order:

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

We give one last algorithm, membership in a trie:

```
(* mem : word → trie → bool *)
value rec mem w = fun
  [ Trie(b,l) → match w with
    [ [] → b
    | [n::r] →
        try mem r (List.assoc n l)
        with [ Not_found → False ]
    ]
  ];
```

These recursive algorithms are fairly straightforward. They are easy to debug, maintain and modify due to the strong typing safeguard of ML, and even easy to formally certify. They are nonetheless efficient enough for production use. As we shall see in the next section, using proper sharing they produce compact enough representations for manipulating lexicons of several hundred thousand entries.

Tries may be considered as deterministic finite state automata graphs for accepting the (finite) language they represent. This remark is the basis for many lexicon processing libraries. Actually, the *mem* algorithm may be seen as an interpreter for such an automaton, taking its state graph as its trie argument, and its input tape as its word one. The boolean information in a trie node indicates whether or not this node represents an accepting state. These automata are not minimal, since while they share initial equivalent states, there is no sharing of accepting paths, for which a refinement of lexical trees into dags is necessary. Let us now look at this problem.

3 Sharing

Sharing data representation is a very general problem. Sharing identical representations is ultimately the responsibility of the runtime system, which allocates and desallocates data with dynamic memory management processes such as garbage collectors.

But sharing of representations of the same type may also be programmed by bottom-up computation. All that is needed is a memo function building the corresponding map without duplications. Let us show the generic algorithm, as an ML *functor*.

3.1 The share functor

This functor (that is, parametric module) takes as parameter an algebra with its domain seen here as an abstract type. Here is its public interface declaration:

```
module Share : functor (Algebra:sig type domain; value size: int; end) →
 sig value share: Algebra.domain → int → Algebra.domain;
end;
```

That is, *Share* takes as argument a module *Algebra* providing a type *domain* and an integer value *size*, and it defines a value *share* of the stated type. We assume that the elements from the domain are presented with an integer key bounded by Algebra.size. That is, *share x k* will assume as precondition that $0 \leqslant k < Max$ with Max =*Algebra.size*.

We shall construct the sharing map with the help of a hash table, made up of pairs $(k, [e_1; e_2; ...e_n])$ where each element e_i in the bucket list has key k.

```
type bucket = list Algebra.domain;

value memo = Array.create Algebra.size ([] : bucket);
```

That is, we create the memory as a hash-table array of a given size and of the right bucket type.

We shall use a service function *search*, such that *search e l* returns the first y in l such that $y = e$ or or else raises the exception *Not_found*.

```
value search e = List.find (fun x → x=e);
```

Now *share x k*, where k is the key of x, looks in k-th bucket l (meaningful since we assume that the key fits in the size: $0 \leqslant k <$*Algebra.size*) and returns y in l such that $y = x$ if it exists, and otherwise returns x memorized in the new k-th bucket $[x :: e]$. Since *share* is the only operation on buckets, we maintain that such y is unique in its bucket when it exists.

```
value share element key =
  let bucket = memo.(key) in
  try search element bucket with
  [Not_found → do {memo.(key):=[element::bucket]; element}];
```

Instead of *share* we could have used the name *recall*, since either we recall a previously archived equal element, or else this element is archived for future recall. It is an interesting property of this modular design that sharing and archiving are abstracted as a common notion.

3.2 Compressing tries

We may for instance instantiate *Share* on the algebra of tries, with a size *hash_max* depending on the application:

```
module Dag = Share (struct type domain=trie;
                           value size=hash_max; end);
```

And now we compress a trie into a minimal dag using *share* by a simple bottom-up traversal, where the key is computed along by hashing. For this we define a general bottom-up traversal function, which applies a parametric *lookup* function to every node and its associated key.

```
value hash0 = 1    (* linear hash−code parameters *)
and hash1 letter key sum = sum + letter×key
and hash b arcs = (arcs + if b then 1 else 0) mod hash_max;

value traverse lookup = travel
 where rec travel = fun
  [ Trie(b,arcs) →
     let f ( tries ,span) (n,t) =
        let (t0,k) = travel t
        in ([( n,t0):: tries ], hash1 n k span)
     in let (arcs0,span) = List. fold_left f ([], hash0) arcs
        in let key = hash b span
           in (lookup (Trie(b,rev arcs0)) key, key)
  ];
```

```
value compress = traverse Dag.share;

value minimize trie = let (dag,_) = compress trie in dag;
```

Despite its simplicity, this algorithm is rather efficient. In our Sanskrit application, the trie of 11500 entries is shrunk from 219KB to 103KB in 0.1s, whereas the trie of 120000 inflected forms is shrunk from 1.63MB to 140KB in 0.5s on a 864MHz PC. A trie of 173528 English words is shrunk from 4.5MB to 1.1MB in 2.7s. Measurements showed that the time complexity is linear with the size of the lexicon (within comparable sets of words). 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 (Flajolet *et al.*, 1990).

Tuning of the hash function parameters leads to many variations. For instance if we assume an infinite memory we may turn the hash calculation into a one-to-one Gödel numbering, and at the opposite end taking *hash_max* to 1 we would do list lookup in the unique bucket, with worse than quadratic performance.

Using hash tables for sharing with bottom-up traversal is a standard dynamic programming technique, but the usual way is to delegate computation of the hash function to some hash library, using a generic low-level package. This is what happens for instance if one uses the module hashtbl from the Ocaml library. Here the *Share* module does *not* compute the keys, which are computed on the client side, avoiding re-exploration of the structures. That is, Share is just an associative memory. Furthermore, key computation may take advantage of specific statistical distribution of the application domain in order to improve bucket search.

3.3 Sharing as minimization

We ramarked above that tries could be seen as deterministic automata graphs for recognizing the language of their lexicon. Such languages being finite, their recognizer state graph is acyclic. Provided the tries are not carrying spurious irrelevant nodes, their dag minimization actually constructs the minimal automaton recognizing their language. Let us make this intuition precise.

Definition
A trie *Trie(b,arcs)* is said to be *reduced* iff either *arcs* = [] or else *arcs* = $[(l_1, t_1); ...(l_n, t_n)]$ with all l_i's distinct, and all the t_i's are reduced and non-empty tries.

Note that tries constructed by *make_lex* are reduced by construction.

Fact 1
Reduced tries are structurally equal iff they have the same contents (Proof left to the reader).

Fact 2
Sharing reduced tries as dags yields the minimum automaton recognizing the langage of their contents.

Proof
Consider a deterministic automaton recognizing the language of the contents of a dagified reduced trie. Every path in the dag maps a node of the trie into a state of the automaton (by following the corresponding sequence of transitions from its initial state). Assume two distinct paths u_1 and u_2 map to the same state σ. Let T_1 and T_2 be the two corresponding subtries. By Fact 1, some word w belongs to one and not to the other. Thus u_1w must be accepted by the automaton and u_2w not accepted (or conversely), a contradiction, since the state obtained from σ by transition w cannot be both accepting and non-accepting. □

We shall see later another application of the *Share* functor to the minimization of the state space of other finite automata, possibly cyclic, possibly non-deterministic, and possibly transducers producing output.

4 Decorated tries for flexed forms storage

4.1 Decorated tries

A set of elements of some type τ may be identified as its characteristic predicate in $\tau \rightarrow$ *bool*. A trie with boolean information may similarly be generalized to a structure representing a map, or function from words to some target type, by storing elements of that type in the information slot.

To distinguish absence of information, we could use a type *(option info)* with constructor *None*, presence of value *v* being indicated by *Some(v)*. We rather choose here a variant with lists, which are versatile to represent sets, feature structures, etc. Now we may associate to a word a non-empty list of information of polymorphic type α, absence of information being encoded by the empty list. We shall call such associations a *decorated trie*, or *deco* in short.

```
type deco α = [ Deco of (list α × darcs α) ]
and darcs α = list (letter × deco α)
```

The zipper type is adapted in the obvious way, and algorithm *zip_up* is unchanged. Function *trie_of* becomes *deco_of*, taking as extra argument the information associated with the singleton trie it constructs:

```
(* deco_of : ( list  α) → word → (deco α) *)
value deco_of i  = decrec
  where rec decrec = fun
    [ [] → Deco(i,[])
    | [n :: rest ] → Deco([],[(n,decrec rest )])
    ];
```

Note how the empty list *[]* codes absence of information. We generalize algorithm *enter* into *add_deco*, which unions new information to previous one:

```
(* add_deco : (deco α) → word → (list α) → (deco α) *)
value add_deco deco word i = enter_edit Top deco word
  where rec  enter_edit  z d = fun
```

```
[ [] → match d with
        [ Deco(j,l) → zip_up z (Deco(union i j,l)) ]
| [n :: rest ] → match d with
    [ Deco(j,l) → let (left, right) = zip n l
                  in match right with
       [ [] → zip_up (Zip(j,left,n,[], z)) (deco_of i rest)
       | [(m,u)::r] →
    if m=n then enter_edit (Zip(j, left ,n,r,z)) u rest
    else zip_up (Zip(j, left ,n, right ,z)) (deco_of i rest)
       ]
    ]
];
```

4.2 Lexical maps

We can easily generalize sharing to decorated tries. However, substantial savings will result only if the information at a given node is a function of the subtrie at that node, i.e. if such information is defined as a *trie morphism*. This will not be generally the case, since this information is in general a function of the word stored at that point, and thus of all the accessing path to that node. The way in which the information is encoded is of course crucial. For instance, encoding morphological derivation as an operation on the suffix of an inflected form is likely to be amenable to sharing common suffixes in the inflected trie, whereas encoding it as an operation on the whole stem will prevent any such sharing.

In order to facilitate the sharing of mappings that preserve an initial prefix of a word, we shall use the notion of *differential word*. A differential word is a notation permitting one to retrieve a word w from another word w' sharing a common prefix, as follows.

```
type delta = (int × word);
```

We compute the difference between w and w' as a *differential word* $(|w_1|, w_2)$ where $w = p.w_1$ and $w' = p.w_2$, with maximal prefix p. In ML, we compute *diff* w w', where:

```
value rec diff = fun
    [ [] → fun x → (0,x)
    | [c :: r] as w → fun
        [ [] → (length w,[])
        | [c' :: r'] as w' →
              if c = c' then diff r r'
              else (length w,w')
        ]
    ];
```

Now w' may be retrieved from w and $d = diff\ w\ w'$ as $w' = patch\ d\ w$, with:

```
value patch (n,w2) w =
  let p=truncate n (rev w) in unstack p w2;
```

where *truncate n w* is a list library function, truncating the initial prefix of length n from a word w.

We may now store inverse maps of lexical relations (such as morphological derivations) using the following types (where the type parameter α codes the relation):

```
type inverse α = (delta × α)
and inverse_map α = list (inverse α);
```

Such inverse relations may be used as decorations of specific decos called *lexical maps*:

```
type lexmap α = deco (inverse α);
```

Typically, if word w is stored at a node *Deco([...; (d,r); ...],...)*, this represents the fact that w is the image by relation r of $w' = patch\ d\ w$. Such a *lexmap* is thus a representation of the image by r of a source lexicon. This representation is invertible, while preserving maximally the sharing of prefixes, and thus being amenable to sharing.

As a typical application of this idea, consider the task of building the dictionary of plural forms in a language such as English. Every regular plural form will be labeled by a differential word of the special form (0, []), meaning "I am the plural of the word stored in my father node". All such forms will be shared, whenever the plural form is not the prefix of some other word.

4.3 Lexicon repositories using tries, decos and lexmaps

In a typical computational linguistics application, grammatical information (part of speech role, gender/number for substantives, valency and other subcategorization information for verbs, etc) may be stored as decoration of the lexicon of roots/stems. From such a decorated trie a morphological processor may compute the lexmap of all inflected forms, decorated with their derivation information encoded as an inverse map. This structure may itself be used by a tagging processor to construct the linear representation of a sentence decorated by feature structures. Such a representation will support further processing, such as computing syntactic and functional structures, typically as solutions of constraint satisfaction problems.

Let us for example give some information on the indexing structures *trie*, *deco* and *lexmap* used in our computational linguistics tools for Sanskrit.

The main linguistic component in our Sanskrit platform is a structured lexical database, from which various documents may be produced mechanically, such as a printable dictionary through a TEX/pdf compiling chain, and a Web site (`http://pauillac.inria.fr/~huet/SKT`) with indexing tools. The index CGI engine searches the words by navigating in a persistent trie index of stem entries. In the current version, the database comprises 12525 items. The corresponding trie (shared as a dag) has a size of 112KB.

When computing this index, another persistent structure is created. It records in a deco all the genders associated with a noun entry (nouns comprise substantives and adjectives, a blurred distinction in Sanskrit). At present, this deco records genders for 7459 nouns, and it has a size of 287KB.

A separate process may then iterate on this genders structure a grammatical engine, which for each stem and associated gender generates all the corresponding declined forms. Sanskrit has a specially prolific morphology, with three genders, three numbers and seven cases. The grammar rules are encoded into 84 declension tables, and for each declension suffix an internal sandhi computation is effected to compute the final inflected form. All such words are recorded in an inflected forms lexmap, which stores for every word the list of pairs (stem,declension) that may produce it. This lexmap records 130885 such inflected forms with associated grammatical information, and it has a size of 363KB (after minimization by sharing, which contracts approximately by a factor of 10). A companion trie, without the information, keeps the index of inflected words as a minimized structure of 154KB.

Similarly, another deco stores information about verbal roots, such as their present class (12KB for 477 roots). A conjugation engine produces 53107 verbal root forms, stored in a lexmap of 973KB. The corresponding trie of inflected words has size 42KB. At the time of writing, the conjugation engine generates the forms for present indicative and passive, imperfect, optative, imperative, perfect and future. Finally, a trie of 26KB holds information about which sequences of preverbs are used for each root.

5 Finite state machines as lexicon morphisms

5.1 Finite-state tradition

Computational phonetics and morphology is one of the main applications of finite state methods: regular expressions, rational languages, finite-state automata and transducers, rational relations have been the topic of systematic investigations (Mohri, 1997; Roche & Schabes, 1997), and have been used widely in speech recognition and natural language processing applications. These methods usually combine logical structures such as rewrite rules with statistical ones such as weighted automata derived from hidden Markov chains analysis in corpuses. In morphology, the pioneering work of Koskenniemi (1984) was put in a systematic framework of rational relations and transducers by the work of Kaplan & Kay (1994) that is the basis for the Xerox morphology toolset (Karttunen, 2000; Karttunen, 1995; Beesley & Karttunen, 2003). In such approaches, lexical data bases and phonetic and morphological transformations are systematically compiled in a low-level algebra of finite-state machines operators. Similar toolsets have been developed at University Paris VII, Groningen University, Bell Labs, Mitsubishi Labs, etc.

Compiling complex rewrite rules in rational transducers is however rather subtle. Some high-level operations are more easily expressed over deterministic automata, other ones are easier to state with ϵ-transitions, still others demand non-deterministic descriptions. Inter-traductions are well known, but tend to make the compiled systems bulky, since for instance removing non-determinism is an exponential operation in the worst case. Knowing when to compact and minimize the descriptions is a craft that is not widely disseminated, and thus there is a gap between theoretical descriptions, widely available, and operational technology, kept confidential.

Here we shall depart from this fine-grained methodology and propose more direct translations that place the lexicon in the center role. The resulting algorithms will not have the full generality of the standard approach, although we shall see that an important family of transducers, implementing non-overlapping rewrite rules, is accommodated in a straightforward fashion.

The point of departure of our approach is the above remark that a lexicon represented as a lexical tree or trie is *directly* the state space representation of the (deterministic) finite state machine that recognizes its words, and that its minimization consists *exactly* in sharing the lexical tree as a dag. Thus we are in a case where the state graph of such finite languages recognizers is an acyclic structure. Such a pure data structure may be easily built without mutable references, and thus may be allocated in the static part of the heap, which the garbage collector need not visit, an essential practical consideration. Furthermore, avoiding a costly reconstruction of the automaton from the lexicon database is a computational advantage.

In the same spirit, we shall define automata that implement non-trivial rational relations (and their inversion) and whose state structure is nonetheless a more or less direct decoration of the lexicon trie.

5.2 Unglueing

We shall start with a toy problem that is the simplest case of juncture analysis, namely when there are no non-trivial juncture rules, and segmentation consists just in retrieving the words of a sentence glued together in one long string of characters (or phonemes). Let us consider an instance of the problem say in written English. You have a text file consisting of a sequence of words separated with blanks, and you have a lexicon complete for this text (for instance, 'spell' has been successfully applied). Now, suppose you make some editing mistake, which removes all spaces, and the task is to undo this operation to restore the original.

We shall show that the corresponding transducer may be defined as a simple navigation in the lexical tree state space, but now with a measure of non-determinism. Let us give the detailed construction of this unglueing automaton.

The transducer is defined as a functor, taking the lexicon trie structure as parameter:

module Unglue (Lexicon:**sig** **value** lexicon : Trie.trie; **end**)

First we define the relevant types and exceptions:

type input = word *(* input phrase as a word code *)*
and output = list word; *(* output is sequence of words *)*

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

exception Finished;
```

Now we define our unglueing reactive engine as a recursive process that navigates directly on the (inflected) lexicon trie (typically the compressed trie resulting from the Dag module considered above). The reactive engine takes as arguments the (remaining) input, the (partially constructed) list of words returned as output, a backtrack stack whose items are (input,output) pairs, the path *occ* in the state graph stacking (the reverse of) the current common prefix of the candidate words, and finally the current trie node as its current state. When the state is accepting, we push it on the backtrack stack, because we want to favor possible longer words, and so we continue reading the input until either we exhaust the input, or the next input character is inconsistent with the lexicon data:

```
value rec react input output back occ = fun
  [ Trie(b,arcs) →
      let continue cont = match input with
          [ [] → backtrack cont
          | [ letter :: rest ] →
              try let next = List.assoc letter arcs
                   in react rest output cont [ letter :: occ] next
              with [ Not_found→ backtrack cont ]
          ]
      in if b then
            let pushout = [occ :: output] in
              if input=[] then (pushout,back) (* solution *)
              else let pushback = [(input,pushout) :: back]
                   in continue pushback
         else continue back
  ]
and backtrack = fun
  [ [] → raise Finished
  | [(input,output) :: back] →
        react input output back [] init
        where init = Lexicon.lexicon
  ];
```

Now, unglueing a sentence is just calling the reactive engine from the appropriate initial backtrack situation:

```
value unglue sentence = backtrack [(sentence ,[])];
```

The method is complete, relatively to the lexicon: if the input sentence may be obtained by glueing words from the lexicon, *unglue sentence* will return one possible solution. For instance, assuming the sentence is French Childish Scatology:

```
module Childtalk = struct
value  lexicon  = make_lex ["boudin"; "caca"; "pipi"];
end;

module Childish = Unglue(Childtalk);
```

Now, calling *Childish.unglue* on the encoding of string *"pipicacaboudin"* produces a pair *(sol,cont)* where the reverse of *sol* is a list of words that, if they are themselves reversed and decoded, yields the expected answer *["pipi"; "caca"; "boudin"]*.

5.3 Resumptions and coroutines: a non-determinism toolkit

Of course there may be several solutions to the unglueing problem, and this is the rationale of the *cont* component, which is a *resumption*. For instance, in the previous example, *cont* is empty, indicating that the solution *sol* is unique.

The process *backtrack* may thus be used in coroutine with a solution printer. Assuming a service routine *print_out* that prints solutions with their rank, we thus define (and this now concludes the module *Unglue*):

```
(* resume : resumption → int → resumption *)
value  resume cont n =
   let  (output,resumption) = backtrack cont
   in  do {  print_out  n output; resumption };

value  unglue_all  sentence  = restore  [(sentence ,[])]  1
   where rec  restore  cont  n =
   try  let  resumption = resume cont n
        in  restore  resumption (n+1)
   with  [  Finished →
             if  n=1 then print "␣No␣solution␣" else () ];
```

Let us test this segmenter to solve an English charade[2]:

```
module Short = struct
value  lexicon  = make_lex
 ["able";"am";"amiable";"get";"her";"i";"to";"together"];
end;

module Charade = Unglue(Short);
```

and we get all solutions to the charade, as a "quatrain polisson":

```
Charade.unglue_all (encode "amiabletogether");
 Solution 1 : amiable together
 Solution 2 : amiable to get her
 Solution 3 : am i able together
 Solution 4 : am i able to get her
```

[2] Borrowed from *"Palindroms and Anagrams"*, Howard W. Bergerson, Dover 1973.

Unglueing is what is needed to segment a language like Chinese. Realistic segmenters for Chinese have actually been built using such finite-state lexicon driven methods, refined by stochastic weightings (Sproat *et al*., 1996).

Several combinatorial problems map to variants of unglueing. For instance, over a one-letter alphabet, we get the Fröbenius problem of finding partitions of integers into given denominations[3]. Here is how to give the change in pennies, nickels and dimes:

```
value rec unary = fun [ 0 → "" | n → "|" ^ (unary (n−1)) ];

let penny  = unary 1
and nickel = unary 5
and dime   = unary 10;

module Coins = struct
value lexicon = Lexicon.make_lex [penny; nickel; dime];
end;

module Frobenius = Unglue(Coins);

value change n = Frobenius.unglue_all (encode (unary n));

change 17;
```

```
 Solution 1 :
|||||||||| ||||| | |
...
 Solution 80 :
| | | | | | | | | | | | | | | | |
```

We remark that coroutine programming is basically trivial in a functional programming language, provided one identifies well the search space, states of computation are stored as pure data structures (which cannot get corrupted by pointer mutation), and fairness is taken care of by a termination argument (here this amounts to proving that *react* always terminate).

The reader will note that the very same state graph that was originally the state space of the deterministic lexicon lookup is used here for a possibly non-deterministic transduction. What changes is not the state space, but the way it is traversed. That is we clearly separate the notion of finite-state graph, a data structure, from the notion of a reactive process, which uses this graph as a component of its computation space, other components being the input and output tapes, possibly a backtrack stack, etc.

We shall continue to investigate transducers that are lexicon mappings, but now with an explicit non-determinism state component. Such components, whose

[3] Except that we get permutations since here the order of coins matters.

structure may vary according to the particular construction, are decorations on the lexicon structure, which is seen as the basic deterministic state skeleton of all processes that are lexicon-driven; we shall say that such processes are *lexicon morphisms* whenever the decoration of a lexicon trie node is a function of the subtrie at that node. This property entails an important efficiency consideration, since the sharing of the trie as a dag may be preserved when constructing the automaton structure:

Fact. Every lexicon morphism may minimize its state space isomorphically with the dag maximal sharing of the lexical tree. That is, we may directly decorate the lexicon dag, since in this case decorations are invariant by subtree sharing.

There are numerous practical applications of this general methodology. For instance, we shall show in section 7 below how to construct a segmenter as a decorated inflected forms lexicon, where the decorations express application of the euphony rules at the juncture between words. This construction is a direct extension of the unglueing construction, which is the special case when there are no euphony rules, or when they are optional.

But first we must explain what exactly we mean by euphony rules.

6 Rewrite rules for reversible transducers

6.1 Phonetics and euphony

The utterance of a phoneme demands a certain configuration of the vocal apparatus: articulation point of the tongue within the mouth, opening or closing of the nasal cavity, vibration or not of the larynx, etc. Uttering a sequence of phonemes provokes physiological transformations and incurs an expense of energy. Minimization of this energy leads to the smoothing of the vocal signal, and its discretization leads to phoneme transformations. When the transformation is local to a word, we speak of *internal sandhi*, a process that transforms the sequence of morphemes from which the word originates into a smoothly euphonic stream of phonemes that stabilises to the *standard* pronunciation of the word in a given state of development of a language. Such transformations are frozen forms, at the time scale of the synchronous view of a language (whereas it may continue to evolve in the diachronous point of view). These transformations may or may not be apparent in the spelling of the word. Thus the voiced *[b]* in the French verb *absorber* becomes the surd *[p]* in the derived substantive *absorption*, whereas in English the *[z]* sound of *dogs* is not distinguished from the *[s]* sound of *cats* in the written form.

Similar phonetic fusion processes occur at the juncture of successive words in a spoken sentence, but such *external sandhi* is usually less permanently marked, and seldom indicated in writing. In French *external sandhi* involves the liaison, its absence with the so-called aspirated *h* leading to hiatus, elisions like in *maître d'hôtel*, and the euphonic *t* in “Malbrough s'en va-t-en guerre”. In Sanskrit however, such euphonic transformations have been systematically studied, standardized in grammar rules, and applied to the written representation, which reflects faithfully the normalized pronounciation. Thus the demonstrative pronoun *tad* (this) followed by the absolutive *śrutvā* (heard) becomes *tacchrutvā* “having heard this”. This merging of sounds is reflected in writing by a contiguous chain of letters[4], further glued together by complex ligatures in one continuous drawing. Thus, in the *devanāgarī* system, we get तद् (*tad*) joined to श्रुत्वा (*śrutvā*) to form तच्छ्रुत्वा (*tacchrutvā*). Retrieving the words within the sentence amounts to our unglueing process above, aggravated by the fact that sandhi must be undone, leading to a complex non-deterministic analysis. It is the solving of this segmentation problem that is the central achievement of the present application.

6.2 *Juncture rewrite rules*

We model external sandhi with rewrite rules of the form $u|v \rightarrow w$, where u, v and w are words (standing for strings of phonemes). Such a rule represents the rational relation that holds between all pairs of strings (from now on we use strings and words interchangeably) $\lambda u|v\rho$ and $\lambda w\rho$, for λ and ρ any strings. The symbol | stands for word juncture. Some rules (terminal sandhi) pertain to the case where u is at the end of a sentence. Using # as the symbol for end of sentence, we may represent them as $u|\# \rightarrow w$, and they represent the relation that holds between $\lambda u|\#$ and λw.

In our application, we shall assume the option rule $\epsilon|\epsilon \rightarrow \epsilon$, making sandhi optional, which has the advantage of avoiding a lot of individual identity rules $u|v \rightarrow uv$ for the cases where there is no transformation (typically between a word ending with a vowel and a word starting with a consonant). This has the advantage that we can use our algorithm alternatively on sandhied or unsandhied text, while it generally does not overgenerate when parsing a sandhied text. However, let us stress that our methodology does not rely on the assumption that the rule replacement is optional, and our algorithms can be adapted easily to the case where this assumption is not met, as indicated in section 8.2. However, it is convenient to expose sandhi in the presence of the option rule, since this last rule glues together words in the precise sense that we studied above, and sandhi analysis will be seen as a direct extension of the unglueing algorithm.

For non-option rules, we shall assume that $u \neq \epsilon$, and that $v = \epsilon$ only for terminal sandhi rules, alleviating the use of the special symbol #. We shall see in the following that we shall have to assume also $w \neq \epsilon$ for non-terminal sandhi rules, in order to ensure termination of our segmenter.

We shall also consider contextual rewrite rules of the form $[x]u|v \rightarrow w$, with x a (left context) string. They generate the relation that holds between $\lambda xu|v\rho$ and $\lambda xw\rho$. Such a rule is of course equivalent to the rule $xu|v \rightarrow xw$, but we shall see that contextual rules are treated in a way that optimizes their computational treatment. Figure 1 shows the juncture of two phonetic words, their smoothing, and the phonemic discretization of the situation with a rewrite rule. This drawing is more a didactic explanation of the physiologico-acoustic process than a scientifically precise representation.

[4] Actually word breaks are allowed at certain positions that depend on syllabic and morphological structure, but this does not concern us here.

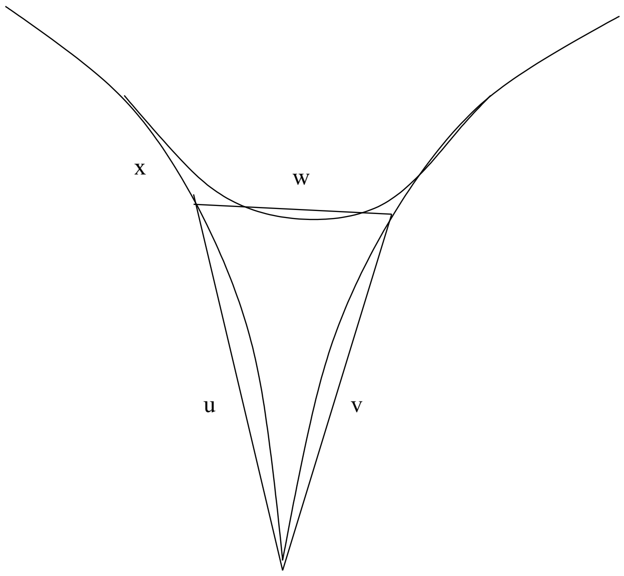

Fig. 1. Juncture euphony and its discretization.

The sandhi problem may then be posed as a regular expression problem, namely the correspondance between $(L \cdot |)^*$ and Σ^* by relation $\mathscr{R}$, where Σ is the word alphabet (not comprising the special symbol |), L is the set of inflected forms, and $\mathscr{R}$ is the rational relation that is the concatenation closure of the union of the rational relations corresponding to the sandhi rules. This presentation is a standard one since the classic work of Kaplan & Kay (1994), and is the basis of the Xerox finite state morphological package (Karttunen, 2000; Karttunen, 1995; Beesley & Karttunen, 2003). In the Kaplan and Kay notation, the rule we write $[x]u|v \rightarrow w$ would be written as $u \text{ --- } v \rightarrow w \,/\, x \,\_\_$. A discussion of the generality of our approach is given in section 10.5.

Note that the sandhi problem is expressed in a symmetric way. Going from $z_1|z_2|...z_n| \in (L \cdot |)^*$ to $s \in \Sigma^*$ is generating a correct phonemic sentence s with word forms $z_1, z_2, ...z_n$, using the sandhi transformations. Whereas going the other way means analysing the sentence s as a possible phonemic stream using words from the lexicon transformed by sandhi. It is this second problem we are interested in solving, since sandhi, while basically deterministic in generation, is strongly ambiguous in analysis.

7 Construction of a segmenting automaton

We shall now use the inflected forms trie as the deterministic skeleton of a non-deterministic finite-state transducer solving the sandhi problem for analysis, by decorating it with rewrite opportunities.

7.1 Choice points compiling from rewriting rules

The algorithm proceeds in one bottom-up sweep of the inflected forms trie. For every accepting node (i.e. lexicon word), at occurrence z, we collect all sandhi rules $\sigma : u|v \rightarrow w$ such that u is a terminal substring of $z : z = \lambda u$ for some λ. When we move up the trie, recursively building the automaton graph, we decorate the node at occurrence λ with a choice point labeled with the sandhi rule. This builds in the automaton the prediction structure for rule σ, at distance u above a matching lexicon word. At interpretation time, when we enter the state corresponding to λ, we shall consider this rule as a possible non-deterministic choice, provided the input tape contains w as an initial substring. If this is the case, we shall then move to the state of the automaton at occurrence v (a precomputation checks that all sandhi rules are plausible in the sense that occurrence v exists in the inflected trie, i.e. there are some words that start with string v). When we take this action, the automaton acts as a transducer, by writing on its output tape the pair (z, σ). Note that we do not need to build a looping state graph structure for the automaton, since all loops are implemented by jumps to a "virtual address" v. This allows us to keep within the paradigm of pure functional programming, with no references and no modifiable data structures.

The treatment of a contextual rule $[x]u|v \rightarrow w$ is similar, we check that $z = \lambda x u$, but the decorated state is now at occurrence λx. In both kinds of rules, the choice point is put at the ancestor of z at distance u. This suggests as implementation to compute at the accepting node z a stack of choice points arranged by the lengths of their left component u. Furthermore, once the matching is done, the context x may be dropped when stacking a contextual rule, since it is no more needed.

Figure 2 illustrates the decoration of the trie by a rule, and the reading of the input tape (along the dotted line) at segmentation time.

The current occurrence z is maintained in a stack argument, as a word *occ* representing the reverse of the access string z. To facilitate matching, our sandhi rules are represented as triples *($\bar{u}$,v,w)* where $\bar{u}$ is the word coding the reverse of string u, so that matching amounts to checking that word $\bar{u}$ is an initial sublist of word *occ*.

7.2 Compiling inflected tries as acyclic transducer state dags

Let us first define the relevant data types. First, the lexicon and euphony rules. The lexicon is a *trie*, obtained from the inflected *deco* by forgetting the morphology information, and sharing as a dag.

```
type lexicon = trie
and rule = (word × word × word);
```

The rule triple *($\bar{u}$, v, w)* represents the string rewrite $u|v \rightarrow w$. Now for the transducer state space:

```
type auto = [ State of (bool × deter × choices) ]
and deter = list ( letter × auto)
and choices = list rule;
```

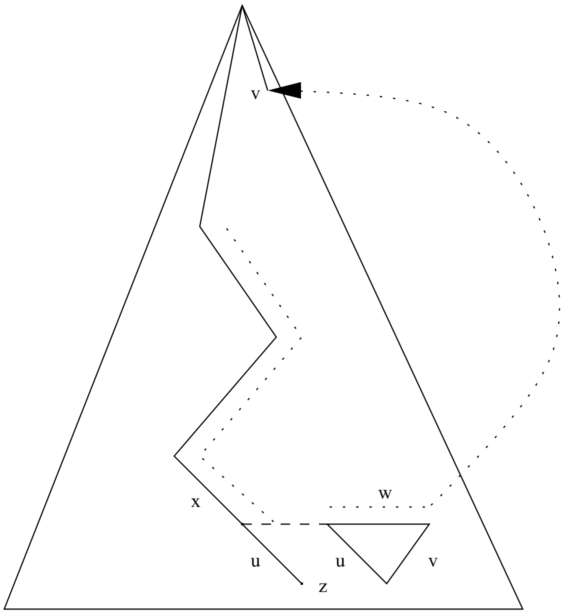

Fig. 2. Decorated lexicon.

The *auto* state *State(b,d,c)* keeps the acceptance boolean *b*, the deterministic skeleton *d*, and the non-deterministic choices *c* given as a list of rules. Note that type *auto* is very similar to type *(deco rule)*, with *deter* playing the role of *(darcs rule)* and *choices* playing the role of *(list rule)*. The only difference is that we keep a boolean information, since choice points label not words in the lexicon, but rather initial subwords where rewriting effect is predicted.

Finally, we stack choice points sets in lists:

```
type stack = list choices;
exception Conflict;
```

We shall minimize our autos at construction time, using exactly the same technology as we used for sharing trees into dags. Our algebra is now the state space:

```
module Auto = Share (struct type domain=auto;
                            value size=hash_max; end);
```

We shall use the simplistic *hash0* and *hash1* hashing primitives already seen, whereas we parameterize *hash* with one extra argument to take care of the rules structure:

```
value hash b arcs rules =
  (arcs + if b then 1 else 0 + length rules) mod hash_max;
```

We are now ready to give the complete ML program that compiles the lexicon index as a transducer, using function *build_auto*:

```
(* build_auto : word → lexicon → (auto × stack × int) *)
value rec build_auto occ = fun
  [ Trie(b,arcs) →
      let local_stack = if b then get_sandhi occ else []
  in let f (deter,stack,span) (n,t) =
      let current = [n :: occ]      (* current occurrence *)
      in let (auto,st,k) = build_auto current t
         in ([(n,auto)::deter],merge st stack,hash1 n k span)
  in let (deter,stack,span) = fold_left f ([],[],hash0) arcs
  in let (h,l) = match stack with
                     [[] → ([],[]) | [h::l] → (h,l)]
  in let key = hash b span h
  in let s = Auto.share (State(b,deter,h)) key
  in (s,merge local_stack l,key)
  ];

(* compile : lexicon → auto *)
value compile lexicon =
  let (transducer,stack,_) = build_auto [] lexicon
  in if stack = [] then transducer else raise Conflict;
```

7.3 Discussion

The most striking feature of this algorithm is its conciseness and efficiency, since the whole computation is done in one linear sweep of the inflected forms trie. We do not give the details of the service function *get_sandhi* that, given word *occ*, returns the matching sandhi rules arranged in a stack $[l_1;l_2;...]$ where l_i is the list of matching rules with $|u| = i$. We do not give either the library function *merge*, which merges such stacks level by level, an easy list programming exercise.

The automaton structure is a tree of nodes *State(b,deter,choices)*, where b is a boolean indicating whether the path from the initial state is an inflected form word, *deter* is its deterministic skeleton, mirroring the structure of the trie of inflected forms, and *choices* is the non-deterministic part, consisting of choice points labeled with euphony rules. These choice points are inserted exactly where the effect of the predicted rule (on an inflected form somewhere below in the deterministic part) starts. *choices* is computed by merging together the stacks of rules computed when constructing its deterministic children. When the current node is created, this stack is popped, and its remainder is merged with the locally matching sandhi rules in order to initialise the choices stack for upper nodes. The main function *compile* checks that at the end of the computation the stack is empty, that is no lexicon item is a proper suffix of some left hand side u of a rewrite rule.

If we had allowed rewrite rules $\sigma : u|v \rightarrow w$ such that $u = \epsilon$, we would have had to provide for *get_sandhi* to return an extra initial layer for such rules, and then modify accordingly function *build_auto* by replacing

```
let (h,l) = match stack with
 ...
in (s,merge local_stack l,key)
```

by:

```
let (h,l) = match (merge local_stack stack) with
 ...
in (s,l,key).
```

A few remarks on state minimization are now in order.

First of all, as the attentive reader will have noticed that there is no analogue in *build_auto* of the reverse operation used in *compress* above. This reversal of arcs was necessary for tries in order to keep the ordering of subtries, since the terminal recursive traversal *fold_left* reverses the order, and since we assume that subtries are given in increasing order of codes of siblings. We have no such invariant in our state space representation, and thus we do not need this reversal. It is assumed that the ordering of siblings, both in the deterministic part and in the non-deterministic part, will be the subject of later optimization, typically by corpus training computing frequency weights. Similarly, if we wanted to optimize lexicon lookup we would have to go back and relax the increasing labels invariant.

Secondly, we remark that it would be incorrect to share states having the same b and d, since the non-deterministic choices substructure may possibly depend on upper nodes because of contextual rules. More precisely, *get_sandhi occ* in *build_auto* will pattern-match a rule $[x]u|v \rightarrow w$ by checking that $z = \lambda xu$, but the decorated auto state is at occurrence λx. That is, the decoration may depend on the path x *above* the decorated lexicon subtrie, and thus *build_auto* is *not* strictly a lexicon morphism in the presence of contextual rules. We see clearly a tension between contextual and non-contextual rules, even though they have the same rewriting power: with contextual rules we get a potentially bigger state space, since some suffix sharing is lost when we compile the lexicon dag. On the other hand, we explore the state space faster using contextual rules: since they label nodes deeper in the tree than the equivalent non-contextual rule, some needless backtrack may be avoided, for solution paths that go through the upper node but not the lower one.

Thirdly, we remark that we arrive at basically the same algorithm for state minimization as the one given in Daciuk *et al.* (2000), but here expressed as a simple application of a generic sharing functor. Furthermore, we obtain a natural minimization algorithm for *non-deterministic* machines, since we represent such machines state spaces as dags. The innovation here is that out of all possible transitions from a state when reading a letter, we favor the one that explores the lexicon structure, as opposed to the phonological transformations. The linguistic rationale is that on average we speak words rather than twist our tongues between them in a sentence.

8 Running the segmenting transducer

8.1 The reactive engine

We assume that we compiled a segmentation transducer from the inflected lexicon trie:

```
value automaton = compile Lexicon.lexicon;
```

The transducer interpreter is a simple reactive engine reading its input tape and making transitions in the automaton state structure, managing the non-deterministic choices with a *resumption* stack and keeping track of its partial output in an *output* stack storing word/transition pairs .

Let us define the various types and exceptions involved.

```
type transition =
  [ Euphony of rule (* (rev u,v,w) st u|v → w *)
  | Id               (* identity or no sandhi *)
  ]
and output = list (word × transition);
```

Similarly to the unglueing situation, z is the reverse of the predicted inflected form, but now it may be paired in the *output* of the transducer with either *Id*, indicating mere glueing (no rewriting), or *Euphony*, indicating non-trivial sandhi. When we backtrack, there are now two situations, one similar to unglueing, when we reach the end of a word, and another one, when non-deterministic sandhi choices exist. In this last case, we stack the list of such choices, together with the current occurrence, needed to construct the partial solution. This gives us a *backtrack* algebra with two constructors:

```
type backtrack =
  [ Next of (input × output × word × choices)
  | Init of (input × output)
  ]
and resumption = list backtrack; (* coroutine resumptions *)

exception Finished;
```

The two *backtrack* constructors correspond to the two kinds of resumptions in the non-deterministic computation. Constructor *Next* indicates a state in which some non-deterministic rewrite choices are still to be explored, whereas *Init* indicates that we have reached the end of a word, and we continue from the initial state of the transducer, assuming absence of sandhi.

Let us now present a few simple service routines. The first one checks the prefix relation between words; the second advances the input tape by n characters; the last accesses the automaton state from its initial state when doing a sandhi transition, using its v part as a virtual address.

```
value rec prefix u v =
  match u with
```

```
    [ [] → True
    | [a::r] → match v with
        [ [] → False
        | [b::s] → a=b && prefix r s
        ]
    ];

(* advance : int → word → word *)
value rec advance n l = if n = 0 then l
                        else advance (n−1) (tl l);

(* access : word → auto *)
value access = acc automaton (* initial state *)
  where rec acc state = fun
   [ [] → state
   | [c::word] → match state with
      [ State(_,deter,_) → acc (List.assoc c deter) word ]
   ];
```

Two things ought to be remarked. The first one is that we assume that the *access* operation will not fail. As said above, this assumption is verified at the time of compiling the sandhi rules: we checked that every rule $\sigma : u|v \rightarrow w$ is relevant in the sense that there exists in the lexicon at least one word starting with v.

The second remark is that access is done in the deterministic part of the automaton: we do not attempt to run through possible non-deterministic choice points. This is justified by the non-cascading nature of external sandhi, we shall come back to this point later.

Let us now present the transducer interpreter. It takes as arguments the input tape represented as a *word*, an accumulator holding the current output (of type *output* given above), the backtrack stack of type *resumption*, the access code in the deterministic part *occ* of type *word* and finally the current transducer state of type *auto*.

```
value rec react input output back occ = fun
  [ State(b,det,choices) →
    (* we try the deterministic space first *)
    let deter cont = match input with
          [ [] → backtrack cont
          | [letter :: rest] →
       try let next_state = List.assoc letter det
           in react rest output cont [letter ::occ] next_state
       with [ Not_found → backtrack cont ]
          ] in
    let cont = if choices=[] then back
               else [Next(input,output,occ,choices)::back]
    in if b then
```

```
          let out = [(occ,Id)::output] (* identity sandhi *)
          in if input=[] then (out,cont) (* solution *)
             else let alterns = [Init(input,out) :: cont]
               (* we first try the longest matching word *)
                  in deter alterns
        else deter cont
  ]
and choose input output back occ = fun
  [ [] → backtrack back
  | [((u,v,w) as rule)::others] →
        let cont = if others=[] then back
                   else [Next(input,output,occ,others) :: back]
        in if prefix w input then
             let tape = advance (length w) input
             and out = [(u @ occ,Euphony(rule))::output]
             in if v=[] (* final sandhi *) then
                  if tape=[] then (out,cont) (* solution *)
                  else backtrack cont
                else let next_state = access v
                     in react tape out cont v next_state
           else backtrack cont
  ]
and backtrack = fun
  [ [] → raise Finished
  | [resume::back] → match resume with
      [ Next(input,output,occ,choices) →
             choose input output back occ choices
      | Init(input,output) →
             react input output back [] automaton
      ]
  ];
```

8.2 Comments and variations

This algorithm is a natural extension of the unglueing reactive engine. When we backtrack, we resume the computation according to the first resumption on the stack; if it is *Next*, we explore the non-deterministic choices with function *choose*; if it is *Init*, we iterate the search by calling *react* from the initial state *automaton*. When the backtrack stack is empty, we raise exception *Finished*.

Function *choose* looks at the current choices list. If it is empty, it backtracks, otherwise it stacks the other alternatives as a *Next* resumption, and checks whether the input is consistent with the right-hand side w of the current *rule*. If it is, we advance the tape accordingly, emit the corresponding transition on the output tape, and jump to the *next_state* by accessing the virtual address v; that is, provided the input tape is not exhausted, in which case we have found a solution if the sandhi rule is final.

In the main routine *react*, we decide to explore the deterministic space before the non-deterministic one, with function *deter*, which attempts to match the input tape with the current lexicon continuations. Thus we stack the non-deterministic *choices* for later consideration with a *Next* resumption. If we have reached an accepting state, that is if we have read a full word from the lexicon, we emit the corresponding transition on the output tape; if the input is exhausted, we have found a potential solution with optional final sandhi. Otherwise, we just stack this partial solution, but first look whether we may recognize a longer word from the lexicon, using *deter*, similarly to the case where the state is not accepting.

For applications where the optional euphony rule $o : \epsilon|\epsilon \rightarrow \epsilon$ is not allowed, the program branch *if b* should be trimmed out, and acceptance would be defined as just finishing the input with a final euphony rule. The boolean component of states is not needed in this case, since the only accepting state is the initial one. This means that a segmenter defined with a complete set of mandatory juncture rules may use as state space just a decorated trie of type *(deco rule)* with no extra information, but the existence of decorations at a certain occurrence in this space has no direct relationship with this occurrence corresponding to a lexicon word.

While it should be clear that the algorithm is complete, since it explores completely the search space by proper management of the backtrack stack, the order of the various choices is arbitrary, in the sense that it does not change the solution set, only the order in which it is enumerated. Here we choose to explore the deterministic space before the non-deterministic one, favoring matching longer words from the lexicon over doing euphony with shorter words. Also, we consider choice points in the order in which they have been computed by the matching algorithm, whereas we could use a more sophisticated algorithm, using priority queues with some frequency count, or some Markov model or other statistical device issued from corpus training. Such refinements are easy to implement as adaptations of our raw basic algorithm.

The full justification of this transducer will be given in section 10, where the well-foundedness of its recursion structure is formally proved, and where we show its correctness and completeness independently from the particular non-deterministic strategy exhibited above.

8.3 The segmenting coroutine

Let us now explain how to use our interpreter as a word segmenter. We enumerate solutions with a resumption manager *resume*, which calls *backtrack* with its resumption argument *cont*, and prints the *n*-th solution with a service routine *print_out*. We omit the details, since this is very similar to what we already saw in section 5.3.

Now, to find a possible segmentation for a sentence, represented as a *word* input, we just invoke *resume* with an *Init* resumption, using the following *segment_one* function, which either returns as a value some pair *(solution, stack) : (output × resumption)*, or else raises the exception *Finished*:

```
value segment_one sentence = resume [Init(sentence ,[])]  1;
```

Similarly, we get all solutions with the following *segment_all* program that just iterates *resume* until *Finished*, in the *unglue_all* style:

```
value segment_all sentence = segment [Init(sentence ,[])]  1
  where rec segment cont n =
  try let resumption = resume cont n
      in segment resumption (n+1)
  with [ Finished →
          if n=1 then print "␣No␣solution␣" else () ];
```

Many variations are of course possible. For instance, the *resume* resumption manager could be used in coroutine fashion with the next phase of parsing, where solutions could be discarded because of lack of chunk agreement or other constraint.

We remark that our resumptions are symbolic descriptions of the part of the search space that is yet to be explored. They are similar to continuations, but note that they are first-order data values: we need neither laziness, nor closures; thus this technique could be implemented directly in a non-functional programming language.

9 Applications to Sanskrit processing

Let us give some sample experiments with our generic morphological toolset applied to Sanskrit.

9.1 Sanskrit segmentation

Let us illustrate our segmenting transducer by giving simple examples of its operation on the Sanskrit reading application. We use in our examples the Velthuis transliteration scheme for representing the *devanāgarī* Sanskrit alphabet. Since verbs are not yet treated, we limit ourselves to noun phrases, a complex enough issue in the presence of arbitrarily nested compounds.

```
value process sentence = segment_all (encode sentence);
```

We first analyse a nominal compound praising Śiva, सुगन्धिंपुष्टिवर्धनम्, and we follow with a small sentence, मार्जारोदुग्धंपिबति (a cat drinks milk).

```
process "sugandhi.mpu.s.tivardhanam";

 Solution 1 :
[  sugandhim with sandhi m|p -> .mp]
[  pu.s.ti with sandhi identity]
[  vardhanam with sandhi identity]

process "maarjaarodugdha.mpibati";

 Solution 1 :
[  maarjaaras with sandhi as|d -> od]
[  dugdham with sandhi m|p -> .mp]
[  pibati with sandhi identity]
```

These easy problems have a unique solution. Longer sentences may overgenerate and yield large numbers of solutions.

9.2 From segmenting to grammatical tagging

Since our segmenter is lexicon-driven, with inflected forms analysis kept in a lexmap indexing every word with its potential lexeme generators, it is easy to combine segmentation and lexicon-lookup in order to refine the segmentation solutions into text tagging with grammatical information, giving for each declined substantive its possible stem, gender, number and case. Let us run again the above examples in this more verbose mode.

```
lemmatize True;

process "sugandhi.mpu.s.tivardhanam";

 Solution 1 :
[  sugandhim
 < { acc. sg. m. }[sugandhi] >  with sandhi m|p -> .mp]
[  pu.s.ti
 < { iic. }[pu.s.ti] >  with sandhi identity]
[  vardhanam
 < { acc. sg. m. | acc. sg. n. | nom. sg. n. | voc. sg. n. }[vardhana] >
   with sandhi identity]

# process "maarjaarodugdha.mpibati";

 Solution 1 :
[  maarjaaras
 < { nom. sg. m. }[maarjaara] >  with sandhi as|d -> od]
[  dugdham
 < { acc. sg. m. | acc. sg. n. | nom. sg. n. | voc. sg. n. }[dugdha] >
   with sandhi m|p -> .mp]
[  pibati
 < { pr. a. sg. 3 }[paa#1] >  with sandhi identity]
```

Thus each solution details for each inflected form segment its possible lemmatizations. We have obtained a grammatical tagger, with two levels of ambiguities: a choice of segment solutions and for each segment a number of lemmatization choices. We have thus paved the way to interaction with a further parsing process, which will examine the plausibility of each solution with regard to constraints such as phrasal agreement or subcategorization by verb valency satisfaction – possibly in cooperation with further semantic levels that may compute distances in ontology classifications of the stems, or statistical information on co-occurrences.

In our Web implementation of this Sanskrit reader[5] the lemma occurrences are direct hyperlinks to the corresponding lexicon entries. Lexicon entries themselves hold grammatical information in the form of hyperlinks to morphology processors, giving a uniform feel of linked linguistic tools "one click away".

So far we assumed that the sandhi rules were modeled as relations on the strings representing the words in contact. Actually, some particular cases of sandhi are more semantic in nature: special rules pertain to the personal pronoun *sa*, and others to substantive declensions in the dual number. We shall deal with these special rules by allowing these sandhis as generally allowed for the corresponding strings, filtering out the extra solutions (such as non-dual declensions that happen to be homophonic to dual declensions) at the tagging stage. This easy resolution of semantic sandhi illustrates the appropriateness of a lexicon-directed methodology.

The problem of recognizing compounds words is specially acute in Sanskrit, since there is no depth limit to such compound chunks – sometimes a full sentence is one giant compound. We deal with this problem by recognizing compounds one piece at a time, using the fact that compound accretion is identical to external sandhi between words. This is indicated in the examples above by the `iic.` notation, standing for *in initio composi*. This puts compound recognition at the level of syntax rather than morphology, a conscious decision to keep morphology finitistic.

At the time of writing, we are able to lemmatize Sanskrit nominal phrases, as well as small sentences with finite verb forms in the final position, in the tenses of the present system (present indicative, imperative, optative and imperfect), reduplicated perfect, future, aorist and present passive. Initial experiments show that the algorithm has to be tuned for short particle words that tend to overgenerate, but the noise ratio seems low enough for the tool to be useful even in the absence of further filtering. Overgeneration occurs also because of verb forms which are theoretically predicted by the grammarians, but which have no attested occurrence in known corpuses. It is expected that (supervised) corpus tuning will suggest trimming strategies (for instance, verbs may use either active or middle voice, but few use both).

A specific difficulty involves the so-called *bahuvrīhi* (much-rice=rich) compounds. Such determinative compounds used as adjectives may admit extra genders in addition to the possible genders of their rightmost segment, and the extra inflected forms have to be accounted for. For instance Śiva's sign (*liṅga*) is a neuter substantive, forming *liṅgam* in the nominative case. But when compounded with *ūrdhva* (upward) it makes *ūrdhvaliṅga* (ithyphallic), typically used as a masculine adjective, yielding an extra form for the nominative case *ūrdhvaliṅgaḥ*. This difficulty is currently handled by keeping track of all such *bahuvrīhi* compounds occurring in the lexicon; a extra pass over the lexicon collects such extra stems, and adds the corresponding inflected forms. This is not fully satisfactory, since we may segment with our reader compounds that are hereditarily generated from root stems, except in the case of *bahuvrīhi* extra genders derivations, for which the full compound must be explicitly present in the lexicon. An alternative solution would be to try and give

[5] Available from http://pauillac.inria.fr/~huet/SKT/Dico/reader.html

all genders to every compound, anticipating every possible *bahuvrīhi* use, at the risk of overgeneration. This extreme measure would be sweeping the problem under the rug anyway, since *bahuvrīhi* semantics is not compositional with compounding in general, and so specific meanings for such compounds must often be listed explicitly in the dictionary. Thus Rāma's father's name Daśaratha "Ten-chariot" does not mean that he possesses 10 chariots, but rather that he is such a powerful monarch that he may drive his war chariot in all directions. Such "frozen" compounds must be accommodated wholesale.

Another difficulty comes from short suffixes such as *-ga*, *-da*, *-pa*, and *-ya*, which make the sandhi analysis grossly overgenerate if treated as compound-forming words. Such derived forms have to be dealt with by the addition of extra morphology paradigms. It is to be expected anyway that the status of derived words, such as the quality substantives (neuters in *-tva* and feminines in *-tā*), the patronyms and other possessive adjectives (obtained by taking the *vṛddhi* vocal degree of the stem with suffix *-ya* or *-ka*), the agent constructions in *-in*, the possessive constructs in *-vat* or *-mat*, etc. will have to be reconsidered, and treated by secondary morphological paradigms. This is after all in conformity with the Pāṇinean tradition and specially the linguistic theory of Patañjali concerning the *taddhita* derivations (Filliozat, 1988).

9.3 Quantitative evaluation

Our functional programming tools are very concise. Yet as executable programs they are reasonably efficient. The complete automaton construction from the inflected forms lexicon takes only 9s on a 864MHz PC. We get a very compact automaton, with only 7337 states, 1438 of which accepting states, fitting in 746KB of memory. Without the sharing, we would have generated about 200000 states for a size of 5.65MB!

Let us give some indications on the nondeterministic structure. The total number of sandhi rules is 2802, of which 2411 are contextual. While 4150 states have no choice points, the remaining 3187 have a non-deterministic component, with a fan-out usually less than 100. The state with worst fan-out concerns the form *parā*, which combines the generative powers of the pronominal adjective *para/parā* with its derivative *parāc* to produce inflected forms *parāk, parāṅ, parāt, parān, parām, parāḥ*, with respective contributions to their parent *parā* of respectively 28, 11, 33, 23, 29 and 40 sandhi choices, totalling 164 potential choices. Fortunately, even in this extreme situation, the actual possible matches against a given input string limit the number of choices to 2; that is, on a given string, there will be at most one backtrack when going through this state, and this is a general situation. Actually, the interpreter is fast enough to appear instantaneous in interactive use on a plain PC.

The heuristic we used to order the solutions is very simple, namely to favor longest matching sequences in the lexicon. The model may be refined into a stochastic algorithm in the usual way, by computing statistical weights by corpus training. An important practical addition that will be needed at that stage will be to make the method robust by allowing recovery in the presence of unknown words. This is an important component of realistic taggers such as Brill's and its successors (Brill, 1992; Roche & Schabes, 1995). A more ambitious extension of this work will be to turn this robustified tagger into an acquisition machinery, in order to bootstrap our simple lexicon into a larger one, complete for a given corpus. This however will force us to face the problem of morphological analysis, in order to propose stems generating an unknown inflected form.

It may come as a surprise that we need so many sandhi rules. For instance, Coulson (1992) describes consonant external sandhi in a one-page grid, with 10 columns for u and 19 raws for v. The first problem is that Coulson uses conditions such as "ended by *ḥ* except *aḥ* and *āḥ*" that we must expand into as many rules as there are letters *a*, *ā*, etc. The second one is that we cannot take advantage of possible factorings according to the value of v, since when compiling the state space we do prediction on the u part but not on the v.

Actually, generating the set of sandhi rules is an interesting challenge, since writing by hand such a large set of rules without mistakes would be hopeless. What we actually did was to represent sandhi by a two-tape automaton, one for the u and one for the v, and to fill sandhi rules tables by systematic evaluation of this automaton for all needed combinations. The two-tape automaton is a formal definition of sandhi that may be compared to traditional definitions such as Coulson's. Details of this compiling process are omitted here.

10 Soundness and completeness of the algorithms

In this last section, we shall formally prove the correctness of our methodology in a general algebraic framework.

10.1 Formalisation

Definitions

A *lexical juncture system* on a finite alphabet Σ is composed of a finite set of words $L \subseteq \Sigma^*$ and a finite set R of rewrite rules of the form $[x]u|v \rightarrow w$, with $x, v, w \in \Sigma^*$ and $u \in \Sigma^+$ ($x = \epsilon$ for non-contextual rules, $v = \epsilon$ for terminal rules). We note R^o for R to which we add the special optional sandhi rule $o : \epsilon|\epsilon \rightarrow \epsilon$.

The word $y \in \Sigma^*$ is said to be *solution* to the system (L, R) iff there exists a sequence $\langle z_1, \sigma_1\rangle ; ...\langle z_p, \sigma_p\rangle$ with $z_j \in L$ and $\sigma_j = [x_j]u_j|v_j \rightarrow w_j \in R^o$ for $(1 \leqslant j \leqslant p)$, $v_p = \epsilon$ and $v_j = \epsilon$ for $j < p$ only if $\sigma_j = o$, subject to the matching conditions: $z_j = v_{j-1}s_jx_ju_j$ for some $s_j \in \Sigma^*$ for all $(1 \leqslant j \leqslant p)$, where by convention $v_0 = \epsilon$, and finally $y = y_1...y_p$ with $y_j = s_jx_jw_j$ for $(1 \leqslant j \leqslant p)$. We also say that such a sequence is an *analysis* of the solution word y.

Let us give a more abstract alternative definition in terms of rational relations.

Definitions

We define the binary relations $\mathscr{R}$ and $\widehat{\mathscr{R}}$ as the inductive closures of the following clauses:

- $xu|v \ \mathscr{R} \ xw$ if $[x]u|v \rightarrow w \in R$ $(v \neq \epsilon)$
- $xu| \ \widehat{\mathscr{R}} \ xw$ if $[x]u|\epsilon \rightarrow w \in R$

- $| \; \mathscr{R} \; \epsilon$
- $| \; \widehat{\mathscr{R}} \; \epsilon$
- $s \; \mathscr{R} \; s$
- $x_1 \; \mathscr{R} \; y_1$ and $x_2 \; \widehat{\mathscr{R}} \; y_2$ imply $x_1x_2 \; \widehat{\mathscr{R}} \; y_1y_2$.

In the clauses above, s, u, v, w, x, y_1, y_2 range over Σ^*, and x_1, x_2 range over $(\Sigma \cup \{|\})^*$, so that $\mathscr{R}, \widehat{\mathscr{R}} \subseteq (\Sigma \cup \{|\})^* \times \Sigma^*$.

Now we say that $s \in \Sigma^*$ is an *(L,R)-sentence* iff there exists $t \in (L \cdot |)^+$ such that $t \; \widehat{\mathscr{R}} \; s$.

It is easy to check that the existence of such t is equivalent to the existence of an analysis showing that s is a solution as defined above. Actually, an analysis gives a precise proof in terms of the inductive clauses above, with $\mathscr{R}$ modelling (parallel disjoint) sandhi and $\widehat{\mathscr{R}}$ modelling (parallel disjoint sandhi followed by) terminal sandhi.

A rewrite rule $\sigma : [x]u|v \rightarrow w$ is said to be *cancelling* iff $v \neq \epsilon$ and $w = \epsilon$. That is, a non-cancelling sandhi rule is allowed to rewrite to the empty string only if it is terminal. The lexical system (L, R) is said to be *strict* if $\epsilon \notin L$ and no rule in R is cancelling.

Finally, we say that (L, R) is *weakly non-overlapping* if there can be no context overlap of juncture rules of R within one word of L. Formally, rules $[x]u|v \rightarrow w$ and $[x']u'|v' \rightarrow w'$ yield a context overlap within $z \in L$ if $z = \lambda xu = v'\rho$ with $|\lambda| < |v'| \leqslant |\lambda x|$.

We shall prove that for weakly non-overlapping strict lexical juncture systems our segmenting algorithm is correct, complete and terminating, in the sense that it returns all solutions in a finite time. The tricky part is to measure the progress of the exploration of the search space by a complexity function χ that defines an appropriate well-founded ordering that decreases during the computation.

10.2 Termination

Definitions

If *res* is a resumption, we define $\chi(res)$ as the multiset of all $\chi(back)$, for *back* a backtrack value in *res*, where $\chi(Next(in, out, occ, ch)) = \langle |in|, |occ|, |ch| \rangle$, and $\chi(Init(in, out)) = \langle |in|, 0, \kappa \rangle$, with $\kappa = 1 + |R|$. κ is chosen in such a way that it exceeds every non-deterministic fan-out of the transducer states.

χ defines a well-founded ordering, with the standard ordering on natural numbers, extended lexicographically to triples for backtrack values and by multiset extension (Dershowitz & Manna, 1979) for resumptions.

We now associate a complexity to every function invocation. First,
$\chi(react\ in\ out\ back\ occ\ state) = \{\langle |in|, |occ|, \kappa \rangle\} \oplus \chi(back)$,
where $\oplus$ is multiset union. Then $\chi(choose\ in\ out\ back\ occ\ ch) = \{\langle |in|, |occ|, |ch| \rangle\} \oplus \chi(back)$. Finally, $\chi(backtrack\ back) = \chi(back)$.

Proposition 1

If the system is strict, every call to $backtrack(cont)$ either raises the exception *Finished*, or else returns a value (out, res) such that $\chi(res) < \chi(cont)$.

Proof
By nœtherian induction over the well-founded ordering computed by χ. It is easy to show that every function invocation decreases the complexity, we leave the details to the reader. $\square$

Corollary
Under the strictness condition, *resume* always terminates, either raising the exception *Finished*, or returning a resumption of lower complexity than its argument. Therefore *segment_all* always terminates with a finite set of solutions.

Strengthening
Since we used a multiset complexity, invariant by permutation of the backtrack values in resumptions, we have actually proved the above results for a more abstract algorithm, where resumptions are not necessarily organized as sequential lists, but may be implemented as priority queues where elements are selected by an unspecified strategy or oracle. Thus these results remain for more sophisticated management policies of non-deterministic choices, obtained for instance by training on some reference annotated corpus.

Necessity of the strictness conditions
If ϵ is in L, a call to *react* will loop, building an infinite analysis attempt iterating (ϵ, o), with o the optional sandhi rule. If the system contains a cancelling rewriting, such as $\sigma : b|a \rightarrow \epsilon$, with $ab \in L$, the segmenter will loop on input a, attempting an infinite analysis iterating (ab, σ). This shows that the strictness condition is necessary for termination.

10.3 Soundness

It remains to show that the returned results of (*segment_all input*) are indeed analyses of *input* in the sense defined above, exhibiting the property for *input* to be a solution to the system in case of success.

We need first to generalize the notion of $y = y_1...y_p$ being a solution to the system, with analysis $z = \langle z_1, \sigma_1\rangle; ...\langle z_p, \sigma_p\rangle$, into a slightly more general notion of partial solution that may be defined inductively. Using the same notations, we do not insist any more that $v_p = \epsilon$, and we then say that $y = y_1...y_p$ is a *partial solution anticipating* v_p. The empty sequence is a partial solution of segment length 0 anticipating ϵ; a partial solution y of segment length p anticipating v_p with analysis z may be extended into a partial solution yy_{p+1} of segment length $p+1$ anticipating v with $z; \langle z_{p+1}, \sigma_{p+1}\rangle$ provided $z_{p+1} \in L$, $\sigma_{p+1} \in R^o$, $z_{p+1} = v_p s_{p+1} x_{p+1} u_{p+1}$ for some $s_{p+1} \in \Sigma^*$, $y_{p+1} = s_{p+1} x_{p+1} w_{p+1}$, and $v = v_{p+1}$. Note that a solution is a partial solution anticipating ϵ.

Proposition 2
Assume the lexical system (L, R) is strict and weakly non-overlapping, and let $s \in \Sigma^*$. We show that every invocation of *react*, *choose* and *backtrack* met in the

computation of *(react s [] [] [] automaton)* enjoys property P defined as follows:
– either its execution raises the exception *Finished*,
– or else it returns a value *(output,cont)* such that *rev(output)* is a valid analysis of s as a solution to (L, R) and *backtrack(cont)* enjoys property P.

Proof
First, we note that the inductive predicate P is well-defined by nœtherian induction on χ, the system being assumed strict. The proof itself is by simultaneous induction, the statement of the proposition being appropriately strengthened for each procedure, as follows. Every tuple $(input, output, occ)$ of values passed as parameters of the invocations or within a backtrack value is such that $s = r \cdot input$ for some $r \in \Sigma^*$ (the already read portion of the input tape), and $rev(output)$ is a valid analysis of r as a partial solution anticipating some prefix of occ. The proof is a routine case analysis, the details of which being left to the reader. We just remark that the proof needs two correctness assumptions on the automaton construction. The first one is that the deterministic structure stores words in L - this follows from the construction of *automaton* by *compile lexicon*. The second one is that its non-deterministic structure is correct with respect to R, that is every *(ū,v,w) as rule* in the *choices* argument of *choose* is such that there exists a rule $[x]u|v \rightarrow w \in R$ with u the reverse of $\bar{u}$, and taking z as the reverse of occ, x is a suffix of z and $z \cdot u \in L$. This property is part of the specification of the service routine *get_sandhi* invoked by *build_auto*. The only tricky part of the proof concerns the case where a contextual rule would fire even though its context is not fully present in the solution. □

Let us see why the non-overlapping condition is necessary to prevent this situation.

Necessity of the non-overlapping condition
Let us consider the juncture system (L, R) with $R = \{\sigma : [b]d| \rightarrow e, \sigma' : a|b \rightarrow c\}$, $L = \{bd, ia\}$. The overlap concerns context b in word bd. The algorithm incorrectly segments the sentence *ice* as [*ia with sandhi* $a|b \rightarrow c$] followed by [*bd with sandhi* $d| \rightarrow e$]; the second rewriting is incorrect since context b is absent from *icd* after application of the first rule.

10.4 Completeness

The segmenting algorithm is not only correct, it is complete:

Proposition 3
Under the same condition of strictness of system (L, R), the segmenting algorithm is complete in the sense that *(segment_all s)* will return all the analyses of s when s is indeed a solution to the system.

This proposition is provable along the same pattern as Proposition 2, of which it is the converse. Actually, the two properties may be proved together within the same induction, every 'if' being strengthened into an 'iff', since it is easy to show that the algorithm covers all possible cases of building a valid partial analysis. This of course requires the corresponding strengthening of the two properties of *build_auto*, namely that the deterministic structure of the automaton is complete for L and that its non-deterministic structure is complete for R. Again we skip the details of the proof, which is straightforward but notationally heavy.

Propositions 1, 2 and 3 may be summed up as:

Theorem
If the lexical system (L, R) is strict and weakly non-overlapping, s is an *(L,R)-sentence* iff the algorithm (*segment_all s*) returns a solution; conversely, the set of all such solutions exhibits all the proofs for s to be an *(L,R)-sentence*.

A variant of the theorem, without the closures $\mid \mathscr{R}\ \epsilon$ and $\mid \widehat{\mathscr{R}}\ \epsilon$ (optional sandhi and terminal sandhi), is obtained by the variant algorithm explained above, where we suppress the program branch *if b* in algorithm *react*. All successes must end with terminal sandhi, and thus the accepting boolean information in the states may be dispensed with. If only certain rules are optional, we may use the obligatory algorithm, complementing every optional rule $[x]u|v \rightarrow w$ with its specific option $[x]u|v \rightarrow uv$.

We note that the weak non-overlapping condition is very mild indeed, since it pertains only to contextual rules. Whenever a contextual rule $[x]u|v \rightarrow w$ forms a context overlap with others, it is enough to replace it with the equivalent non-contextual rule $xu|v \rightarrow xw$ in order to correct the problem. Note that non-contextual rules may have arbitrary overlappings, since we do not cascade replacements (i.e. we do not close our rational relations with transitivity), and thus a juncture rewrite can neither prevent nor help its neighbourgs.

Actually, in practice a stronger non-overlapping condition is met.

Definition
(L, R) is *strongly non-overlapping* if there can be no overlap of juncture rules of R within one word of L. Formally, rules $[x]u|v \rightarrow w$ and $[x']u'|v' \rightarrow w'$ overlap within z if $z = \lambda xu = v'\rho$ with $|\lambda| < |v'|$.

This condition means that the juncture euphony between two words is not disturbed by the previously spoken phoneme stream. We believe that this is a mild condition on the adequation of the euphony system. An overlap would signify that some word is too short to be stable in speech, to the point that it deserves to disappear as an independant lexical item. Indeed, it is the case that:

Fact
In classical Sanskrit, external sandhi is strongly non-overlapping.

This fact is easy to check, since for external sandhi the maximal length of u, v, and x is 1, so we have only to check for words of length at most 2. The only problematic case is the preverb *ā* ("toward"). We accommodate it by keeping the corresponding forms in the inflected lexicon, as opposed to letting the particle overgenerate at the level of external sandhi. This necessitates however a special treatment with a notion of *phantom phoneme*, in order to keep left-associativity of sandhi. We do not develop this further in the present paper, and refer the interested reader to (Huet, 2003b), which explains how to represent preverbs. In the Vedic language, the emphatic particle *u* (indeed, furthermore, now) would also be problematic, although it seems to appear mostly at the end of verses.

In contrast, internal sandhi cascades over morphemes within one word with complex retroflexions, and is not directly amenable to our euphony treatment. Obviously morphology must be treated by a layer of phonetic transformations isolated from the juncture adjustments.

We end this section by remarking that the non-overlapping conditions considered above are not imposing some kind of determinism on juncture rewriting, such as confluence of the corresponding string rewriting system. Indeed they do not rule out ambiguities of application arising from speech variants, such as two rules with same patterns u and v, but distinct replacements w_1 and w_2.

10.5 Comparison with related work

We considered in this work only a simple case of general rational relations as studied by Kaplan & Kay (1994), or even of the replace operator proposed by Karttunen (1995). Our relations are binary, not n-ary. We allow context only to the left. We consider only two relations (sandhi and terminal sandhi), with possibly optional rules. We consider closure by concatenation, yielding one-step parallel replacement, but have not studied complex strategies iterating possibly overlapping replacements. For instance, it is not clear to us how to model *internal* sandhi by cascading regular replacements – thus we are able to compute inflected forms with a specific internal sandhi synthesis procedure, but we do not have an inverse internal sandhi analyzer; such an analyzer would be useful for stemming purposes, by proposing new lemmas for lexicon completion from unknown inflected forms encountered in a corpus. Some hints on how to treat internal sandhi by finite transducers are given in Chapter 3 of Sproat (1992).

Our methodology is close in spirit to Koskenniemi's two-level rules: our segmenter is tightly controlled by matching the lexicon items with the surface form stream, the sandhi rules giving simultaneous constraints on both ends. It is probably within a general two-level regular relations processing system that this segmenting algorithm would better fit (Karttunen & Beesley, 2001).

11 Conclusion

We have exhibited a consistent design for computational morphology mixing lexicon structures and finite automata state space representations within a uniform notion of lexical tree decorated with information structures. These representations are finitely generated structures, which are definable in purely applicative kernels of programming languages, and thus benefit from safety (immutability due to absence of references), ease of formal reasoning (induction principles) and efficiency (static memory allocation). Being acyclic, they may be compressed optimally as dags by a uniform sharing functor. In particular, decorated structures that are lexicon morphisms preserve the natural sharing of the lexicon trie.

An an instance of application, we showed how euphony analysis, inverting rational juncture rewrite rules, was amenable to processing with finite state transducers organized as deterministic lexical automata decorated with non-deterministic choice points predicting euphony. Under a mild assumption of non-interference of euphony rules across words, we showed that the resulting transduction coroutine produced a finite but complete set of solutions to the problem of segmentation of a stream of phonemes modulo euphony.

We showed application of this technique to a lexicon-driven Sanskrit segmenter, resulting in a non-deterministic tagger, complete with respect to the lexicon. Compound analysis from root stems is solved by the same process. We believe this is the first computational solution to *sandhi* analysis. This prototype tagger has been tested satisfactorily on nominal phrases and small sentences. It constitutes the first layer of a Sanskrit processing workbench under development by the author.

This design has been presented as an operational set of programs in the Objective Caml language, providing a free toolkit for morphology experiments, much in the spirit of the Grammatical Framework type theory implementation of Aarne Ranta (Ranta, 2003). This toolkit and its documentation may be freely downloaded from site `http://pauillac.inria.fr/~huet/ZEN/`. This toolkit has been applied by Sylvain Pogodalla and Nicolas Barth to the morphological analysis of French verbs (300 000 inflected forms for 6500 verbs); see `http://www.loria.fr/equipes/calligramme/litote/demos/verbes.html`. Some of the Zen concepts have been reused in the Grammatical Framework implementation.

A systematic applicative representation of finite state machines using the ideas of the Zen toolkit is sketched in Huet (2003a).

References

Beesley, K. R. and Karttunen, L. (2003) *Finite-state morphology*. CSLI Publications, The University of Chicago Press.

Bentley, J. L. and Sedgewick, R. (1997) Fast algorithms for sorting and searching strings. *Proceedings 8th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM.

Bergaigne, A. (1884) *Manuel pour étudier la langue sanscrite*. F. Vieweg, Paris.

Brill, E. (1992) A simple rule-based part of speech tagger. *Proceedings Third Conference on Applied Natural Language Processing*, pp. 152–155.

Burstall, R. (1969) Proving properties of programs by structural induction. *Comput. J.* **12**(1), 41–48.

Burstall, R. (1984) Programming with modules as typed functional programming. *Proc. Int. Conf. on Fifth Gen. Computing systems.*

Coulson, M. (1992) *Sanskrit – an introduction to the classical language*, 2nd ed. Hodder & Stoughton.

Cousineau, G. and Mauny, M. (1998) *The Functional Approach to Programming*. Cambridge University Press.

Daciuk, J., Mihov, S., Watson, B. W. and Watson, R. E. (2000) Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, **26**(1).

de Rauglaudre, D. (2001) *The Camlp4 preprocessor*. `http://caml.inria.fr/camlp4/`

Dershowitz, N. and Manna, Z. (1979) Proving termination with multiset ordering. *Comm. ACM*, **22**, 465–476.

Filliozat, P.-S. (1988) *Grammaire sanskrite pâninéenne*. Picard, Paris.

Flajolet, P., Sipala, P. and Steyaert, J.-M. (1990) Analytic variations on the common subexpresssion problem. *Proceedings of 17th ICALP colloquium*, pp. 220–234. Warwick.

Gordon, M., Milner, R. and Wadsworth, C. (1977) *A metalanguage for interactive proof in LCF*. Technical report, Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh.

Huet, G. (1997) The Zipper. *J. Funct. Program.* **7**(5), 549–554.

Huet, G. (2002) *The Zen computational linguistics toolkit*. Technical report, ESSLLI Course Notes. `http://pauillac.inria.fr/~huet/ZEN/zen.pdf`

Huet, G. (2003a) Automata mista. In: Dershowitz, N. (ed.), *Festschrift in honor of Zohar Manna for his 64th anniversary: Lecture Notes in Computer Science 2772*. Springer-Verlag. `http://pauillac.inria.fr/~huet/PUBLIC/zohar.pdf`

Huet, G. (2003b) Lexicon-directed segmentation and tagging of Sanskrit. *XIIth World Sanskrit Conference*, Helsinki, Finland.

Huet, G. (2003c) Linear contexts and the sharing functor: Techniques for symbolic computation. In: Kamareddine, F. (ed., *Thirty-five Years of Automating Mathematics*. Kluwer.

Huet, G. (2003d) Towards computational processing of Sanskrit. *International Conference on Natural Language Processing (ICON)*, Mysore, Karnataka.

Huet, G. (2003e) Zen and the art of symbolic computing: Light and fast applicative algorithms for computational linguistics. *Practical Aspects of Declarative Languages (PADL) Symposium*. `http://pauillac.inria.fr/~huet/PUBLIC/padl.pdf`

Kaplan, R. M. and Kay, M. (1994) Regular models of phonological rule systems. *Computational Linguistics*, **20**(3), 331–378.

Karttunen, L. (1995) The replace operator. *ACL'95*.

Karttunen, L. (2000) Applications of finite-state transducers in natural language processing. *Proceedings CIAA-2000*.

Karttunen, L. and Beesley, K. R. (2001) A short history of two-level morphology. *ESSLLI'2001 Workshop on Twenty Years of Finite-state Morphology*. `http://www2.parc.com/istl/members/karttune/`

Kessler, B. (1992) *External sandhi in classical Sanskrit*. MPhil thesis, Stanford University.

Kessler, B. (1995) Sandhi and syllables in classical Sanskrit. In: Duncan, E., Farkas, D. and Spaelty, P. (eds.), *Twelfth West Coast Conference on Formal Linguistics*. CSLI.

Koskenniemi, K. (1984) A general computational model for word-form recognition and production. *10th International Conference on Computational Linguistics*.

Landin, P. (1966) The next 700 programming languages. *Comm. ACM*, **9**(3), 157–166.

Laporte, E. (1995) *Rational transductions for phonetic conversion and phonology*. Technical report, IGM 96-14, Institut Gaspard Monge, Université de Marne-la-Vallée.

Leroy, X., Rémy, D., Vouillon, J. and Doligez, D. (2000) *The Objective Caml system, documentation and user's manual – release 3.00*. INRIA. `http://caml.inria.fr/ocaml/`

Mohri, M. (1997) Finite-state transducers in language and speech processing. *Computational Linguistics*, **23**(2), 269–311.

Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

Ranta, A. (2003) Grammatical framework: a type-theoretical grammar formalism. *J. Funct. Program.* **13**.

Roche, E. and Schabes, Y. (1995) Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, **21**(2), 227–253.

Roche, E. and Schabes, Y. (1997) *Finite-state Language Processing*. MIT Press.

Sproat, R. (1992) *Morphology and Computation*. MIT Press.

Sproat, R., Shih, C., Gale, W. and Chang, N. (1996) A stochastic finite-state word-segmentation algorithm for Chinese. *Computational Linguistics*, **22**(3), 377–408.

Weis, P. and Leroy, X. (1999) *Le langage Caml*, 2nd ed. Dunod.

Whitney, W. D. (1924) *Sanskrit grammar*, 5th ed. Leipzig.

Whitney, W. D. (1997) *Roots, verb-forms and primary derivatives of the sanskrit language*. Motilal Banarsidass, Delhi. (1st edition 1885).