Multi-tape Automata for Speech and Language Systems: A Prolog Implementation

Article · October 1998

Source: CiteSeer

CITATIONS
0

READS
28

2 authors, including:

George Kiraz
Princeton University
38 PUBLICATIONS **504** CITATIONS

SEE PROFILE

Multi-tape Automata for Speech and Language Systems: A Prolog Implementation

George Anton Kiraz[1] and Edmund Grimley-Evans[2]

[1] Bell Laboratories, Lucent Technologies
700 Mountain Ave.
Murray Hill, NJ 07974
[2] Computer Laboratory, University of Cambridge
Pembroke Str
Cambridge CB2 3QG

Abstract. This paper describes a Prolog implementation of multi-tape finite-state automata and illustrates its use with a rewrite rules system. Operators which are multi-tape specific are defined and algorithms for constructing their behaviour into multi-tape machines are given.

1 Introduction

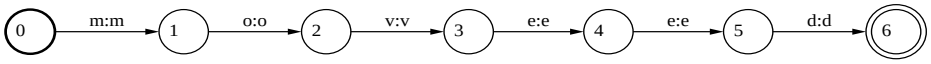
Finite-state machines have been used in natural language and speech systems for a few decades (for the latest in the field, see (Roche, 1997)). Their ease of implementation and mathematical elegance made them popular for many purposes, especially systems which require regular rewrite rules. Traditionally, two versions of finite-state machines have been used: **acceptors** – with one input tape – where each transition is marked with a symbol, and **transducers** (or Mealy machines (Mealy, 1955)) – with one input tape and one output tape – where a transition is marked with two symbols for input and output, respectively. Hence, a transducer is seen as describing a mathematical set-theoretic *binary* relation.

Speech and natural language systems make use of transducers to describe binary relations between two linguistic representations. In morphology, for example, transducers are used with one tape designating **underlying forms** (also called **lexical forms**), while the other describing **surface forms**. Fig. 1(a) gives the identity transducer for English *move* followed by the suffix *ed*. However, the *e* of *move* should be deleted in *moved*. This deletion rule is represented by the transducer in Fig. 1(b). The machine ensures the deletion of *e* (between states 1 and 3, where `Eps` represents ϵ) when it is preceded by *v* (between states 0 and 1) and followed by *ed* (states 3, 5, 0). (States 2 and 4 ensure that if *e* is not deleted, then what follows *cannot* be the suffix *ed*.) The composition of the two machines, shown in Fig. 1(c), gives a new machine which maps *moveed* into *moved*.

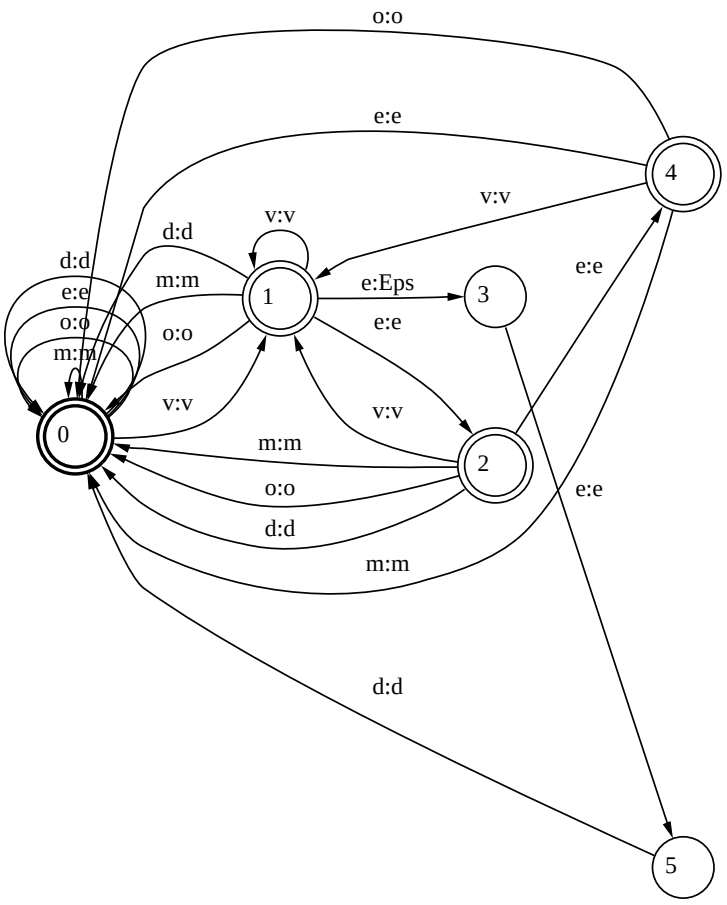
Problems in Semitic morphology require the mapping of *multiple* lexical forms to a surface form. For example, the Syriac[3] surface form *ktab* 'wrote' corresponds to three lexical forms: the root *ktb* which gives the notion of writing, the vowel *a*

[3] Originally an Aramaic dialect, Syriac is a Semitic language with literature spanning

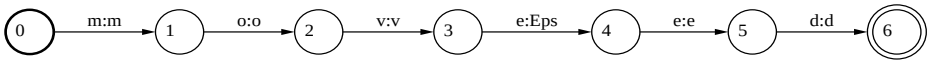
Fig. 1. ϵ-deletion in *moved*



(a) Transducer A: *move+ed*



(b) Transducer B: ϵ-Deletion



(c) Transducer $A \circ B$: *moved*

which indicates past tense, and the some what abstract lexical form *ccvc* (where *c* represents consonants and *v* represents vowels) which indicates the verbal pattern in question. The surface form is derived by substituting the *c*s of the pattern with the letters of the root *ktb* and the *v*s with the vowel *a* resulting in the surface form *ktab*. This phenomenon prompted (Kay, 1987) to propose a multi-tape solution with four tapes: three lexical tapes representing *ktb*, *a* and *ccvc*, respectively, and the fourth surface tape representing *ktab*. Subsequently, (Kiraz, 1994, et seq.) proposed a multi-tape morphology model for which the current Prolog implementation was written.

Detailed examples of using multi-tape automata for this purpose can be found elsewhere (for the latest account, see (Kiraz, Forthcoming)). This paper concentrates on the implementational aspects of the system at hand with an illustration.

Section 2 gives the formal definition of multi-tape automata and introduces regaular relations. Section 3 describes our implementation. Section 4 defines a number of multi-tape operators and shows how to construct multi-tape automata for them. Section 5 gives an evaluation of the system. Section 6 illustrates how the implementation can be used in compiling rewrite rules into automata. Finally, section 7 gives concluding remarks.

2 Multi-tape Automata and Regular Relations

2.1 Definition

Multi-tape finite state machines were first described by (Rabin and Scott, 1959) and (Elgot and Mezei, 1965). Here, we adopt a definition of multi-tape machines based on traditional transducers. An n-**tape finite-state automaton** is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of **states**, Σ is a finite **input alphabet**, δ is a **transition function** mapping $Q \times \Sigma^\epsilon \times \cdots \times \Sigma^\epsilon$ to Q (where $\Sigma^\epsilon = \Sigma \cup \{ \epsilon \}$ and ϵ is the empty string), $q_0 \in Q$ is an **initial state**, and $F \subseteq Q$ is a set of **final states**.

An n-tape FSA accepts an n-tuple of strings if and only if starting from the initial state q_0, it can scan all the symbols on every tape $i, 1 \leq i \leq n$, and end up in a final state $q \in F$.

An n-**tape finite-state transducer** is simply an n-tape finite state automaton but with each tape marked with the word `domain` or `range`; note, however, that transducers are bidirectional.

An ϵ-free FST is an FST which does not have any ϵ-containing transitions. In other words, the transition function δ maps $Q \times \Sigma \times \cdots \times \Sigma$ to Q.

---

from the second century till this day. It is used today by a Christian minority group as a liturgical language in the Middle East, South India (Kerala) and the diaspora (Europe, the Americas and Australia).

2.2 Regular Relations

Parallel to regular languages in formal language theory, regular relations[4] are families of string relations over some alphabet.

String relations are particular collections of ordered tuples of strings over some alphabet Σ, i.e. subsets of $\Sigma^* \times \ldots \times \Sigma^*$. If $X = \langle x_1, x_2, \ldots, x_n \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ are n-tuples of strings, their **n-way concatenation** XY is defined by

$$XY =_{df} \langle x_1 y_1, x_2 y_2, \ldots, x_n y_n \rangle$$

The tuple $X = \langle \epsilon, \ldots, \epsilon \rangle$ is the identity for n-way concatenation.

Based on this, **regular n-relations** are defined along the lines of the recursive definition of regular languages in terms of n-way concatenation (superscript i denotes concatenation repeated i times):

1. The empty set and $\{a\}$ for all $a \in \Sigma^\epsilon \times \ldots \times \Sigma^\epsilon$ are regular n-relations.
2. If R_1, R_2 and R are regular n-relations, then so are
   $R_1 R_2 = \{xy \mid x \in R_1, y \in R_2\}$ (n-way concatenation)
   $R_1 \cup R_2$ (union)
   $R^* = \bigcup_{i=0}^{\infty} R^i$ (Kleene closure)
3. There are no other regular n-relations.

(Kaplan and Kay, 1994) extend the notion of regular expressions to the n-dimensional case. An **n-way regular expression** is a regular expression whose terms are n-tuples of alphabetic symbols or the empty string ϵ. The Kleene correspondence theorem which shows the equivalence between regular n-relations, n-way regular expressions and n-tape finite-state machines is as follows:

1. Every n-way regular expression describes a regular n-relation.
2. Every regular n-relation is described by an n-way regular expression.
3. Every n-tape finite-state transducer accepts a regular n-relation.
4. Every regular n-relation is accepted by an n-tape finite-state transducer.

(Kaplan and Kay, 1994) also define a **same-length regular n-relation** to be a regular n-relation that contains only n-tuples of strings $\langle x_1, \cdots, x_n \rangle$ such that $|x_i| = |x_j|$ for all $1 \leq i, j \leq n$ (all elements of the tuple have the same length). (Kaplan and Kay, 1994) show that R is a same-length regular n-relation if and only if it is accepted by an ϵ-free FST. Further, they demonstrate that regular n-relations are closed under composition, whilst the subclass of same-length regular n-relations is closed under intersection and complementation.

[4] Also called 'transductions' by (Elgot and Mezei, 1965; Nivat, 1968) and 'rational' relations by (Eilenberg, 1974) and other earlier writers. The computational linguistics community uses the term 'regular' relations (Kaplan and Kay, 1994). In what follows, we base our discussion on (Kaplan and Kay, 1994).

3 A Finite-State Calculus

For practical work with finite-state automata, it is good to have a set of general-purpose programs or procedures for manipulating them and carrying out the standard operations such as intersection, union, difference, concatenation, determinisation and minimisation. A comprehensive library of such procedures is often called a finite-state toolkit but we have used the expression 'finite-state calculus' to emphasise our preference for thinking slightly more abstractly in terms of regular relations rather than the concrete automata which represent them: we have usually accessed all functions via a single interface procedure that takes as its input expressions written in an algebraic notation.

3.1 Design

Our choice of Prolog was a design rather than an implementation decision. Although Prolog is less efficient than C in terms of CPU time and memory use – but only by a constant factor! – it is easily portable and has the advantage of being easy to interface with experimental or prototype software for natural language processing (NLP) which is itself more likely to be written in Prolog than in C. We have exploited features of Prolog to make the system particularly suitable for development and prototyping. Some of the features of the system that make it convenient for these purposes are outlined in the following paragraphs.

An Algebraic Interface Suppose one wishes to build an automaton to recognise strings that contain neither adjacent instances of a nor adjacent instances of b. Given primitive procedures `kleene/2`, `intersect/3`, ..., one could achieve this with an explicit sequence of these basic operations. Mathematically, however, one would probably prefer just to write:

$$\overline{\Sigma^* a a \Sigma^*} \cap \overline{\Sigma^* b b \Sigma^*}$$

Fortunately, Prolog allows the user to define an operator with its precedence and type (associativity) and has symbol-processing and matching capabilities (unification) that make it straightforward to use a notation analagous to the mathematical one as an interface to the finite-state calculus. In one of the implementations, this automaton could be constructed with the following procedure call:

```
regexp_to_fsa( (-(($ *)^s(a)^s(a)^($ *))) /\
               (-(($ *)^s(b)^s(b)^($ *))) , answer ).
```

Here `$` replaces Σ as a wild card representing any single terminal symbol, `s(X)` introduces a terminal symbol `X`, `^` represents concatenation, unary '`-`' represents complementation, and so on.

No Declaration or Initialisation Some finite-state toolkits require the user to specify the alphabet in advance. Though this can make it easier to provide a more efficient implementation, it is somewhat inconvenient for experimentation.

For example, the expression above is evaluated correctly without the alphabet having been specified and the resulting automaton will accept the strings *ababa* and *babab*, but also *abcde*, *cccc*, etc. If one does require the alphabet to be limited, which is often the case, one can specify it afterwards by intersecting with the set of strings over that alphabet, for example:

```
regexp_to_fsa( (#answer)/\((s(a)+s(b)+s(c)) *) , answer ).
```

(The prefix operator `#` is used to fetch the automaton previously assigned to that name. The infix operator `+` is for union. Here the result of the intersection is assigned to the same name, replacing the previous value.)

Algebraic Closure The expression 'algebraic closure' is meant to summarise the property that all operations share a domain and range that are the same set. This way any operation may be carried out on any arguments and mathematical identities such as

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

remain universally true in the computational implementation. So

```
regexp_to_fsa( -((#a)/\(#b)) , x ).
```

and

```
regexp_to_fsa( (-(#a))\/(-(#b)) , x ).
```

give the same result whatever the values (or alphabets) of `a` and `b`.

Any Prolog Term to Label a Transition Since Prolog allows terms and subterms to be variables, it is quite natural to want to exploit this in labelling transitions in automata. A trivial example is using the anonymous variable `_` instead of `$` as a wild card, but there are also much more interesting examples. For instance, one can use Prolog terms of the form `t(A,B,C)` to represent 3-tuples of symbols in a same-length multi-tape automaton. Then `t(x,y,_)` represents a 3-tuple with `x` on the first tape, `y` on the second tape and anything on the third tape, and `t(A,_,A)` represents a 3-tuple in which the symbols on the first and third tapes must be the same. (Note that that variables are unique to each transition so it is not possible to use the same trick to force unification between different symbols in a string; that would no longer be finite-state.)

We can also use use the infix operator '`:`' for the special case of 2-tape automata and define a procedure that interprets such automata as transducers and applies them to a 1-tape automaton. An example is:

```
transduce( answer , (V:V+a:b)* , new_answer ).
```

The pair `V:V` maps any symbol to itself and the pair `a:b` maps `a` to `b`, so the transducer `(V:V+a:b)*` copies a string with `a` optionally replaced by `b` throughout.

3.2 Implementation

We have experimented with a number of implementations written entirely in Prolog. The algorithms for the basic operations are standard and can be found, for example, in (Hopcroft and Ullman, 1979), so this section describes only some of the more interesting implementational details.

Representation of Transitions It is not difficult to implement the standard automaton operations, including complementation, without a prior declaration of the alphabet: one can allow some transitions to be labelled with a distinguished symbol 'OTHER' meaning that the transition may be followed on any symbol that is not mentioned on another transition *from the same state*. Some care is required to get a correct implementation. For example, whereas in an ordinary automaton any non-final state that has no transitions leaving it is 'dead' and can be removed, when 'OTHER' transitions are in use it may be necessary to conserve one such 'dead state' in order to keep the transitions to that state whose labels should not be accepted by an 'OTHER' transition.

Unfortunately, when we want to allow transitions to be labelled with arbitrary, non-ground terms, 'OTHER' transitions no longer suffice to represent the result of all finite-state operations. For example, consider the same-length regular relation `(V:V+x:y-x:x)*`, which represents a transducer that compulsorily replaces `x` by `y`. There is no way using 'OTHER' transitions of saying: 'You may follow this transition on any symbol that matches `V:V` *except* symbols that match `x:x`.'

An early implementation of the calculus used 'OTHER' transitions and accepted non-ground terms where possible, testing for the conditions that make some operations impossible. So, if presented with an expression like the one just mentioned, it would print out an apology and fail. Obviously this contradicts the design goal of 'algebraic closure' so a subsequent implementation used 'decision lists' instead.

A 'decision list' is a list of terms of the form `Test-ActionList` interpreted as follows: a fully instantiated Prolog term representing an input symbol is tested agains each `Test` in turn and a transition is made to one of the states listed in the `ActionList` corresponding to the first `Test` that matches; subsequent tests that may match are not considered.

Using a decision list to represent the set of transitions from each state it is possible to represent any finite-state language that can result from the finite-state operations applied to automata in which transitions are labelled with arbitrary Prolog terms.

Importance of Hashing The fastest known algorithm for minimising a finite-state automaton is due to Hopcroft and has running time bounded by $n \log n$ where n is the number of states (and the alphabet is kept constant). A good description of Hopcroft's algorithm can be found in (Watson, 1995), p. 207. However, care is required to convert this description into an implementation that has

the advertised running time. In particular, data structures must be carefully designed to avoid the necessity of conducting linear searches, as for large automata any such linear search will dominate the overall running time. In the case of our Prolog implementations this was achieved by carefully exploiting the way Prolog indexes on the first argument. (When the first argument of a procedure call in instantiated Prolog applies hashing to obtain the same behaviour that is independently specified by the logic, but faster.)

Unfortunately, the implementation that allows arbitrary Prolog terms as labels on transitions does not minimise in time proportional to $n \log n$, though it is usable for automata with several hundred states. In principle $n \log n$ time complexity could be achieved for such automata by compiling the arbitrary Prolog terms into atomic representatives.

Generic Procedures To make the code simpler and easier to check for correctness, generic (or 'meta') procedures, as described for example in (O'Keefe, 1990), p. 246, were used for building automata. So automata may be constructed using a single generic procedure

```
make_automaton(:TransitionPred,:EpsilonPred,:FinalPred,
               +InitialStates,-Automaton)
```

where the arguments `TransitionPred`, `EpsilonPred` and `FinalPred` are the names of procedures that define an automaton by returning the set of labelled transitions from a state, the set of epsilon transitions from a state, and 0 or 1 depending on whether a state is non-final or final, respectively. This technique gave code that was concise and easy to maintain.

Cyclic Structures One implementation of the calculus, instead of numbering the states, represented a state as a list of the transitions from it, where a transition is a pair of label and destination state and the destination state itself is just a (reference to a) similar list of transitions. This way a complete automaton is represented by a single, huge Prolog term that may contain cycles.

Unfortunately, support for cyclic terms is not a standard part of Prolog. SICStus Prolog can 'unify, compare [...], assert, and copy cyclic terms without looping' (Swe, 1995) and the finite-state calculus using cyclic terms did initially appear to work with this Prolog. However, it gave incorrect results with some more complicated examples and the problem was traced to Prolog's term comparison, which turns out not to be well defined for cyclic terms (Carlsson, 1996). Without the standard ordering of terms, it is difficult to manipulate sets of terms: in particular, the standard ordering is used by Prolog's built-in predicate `setof/3`. Because of these practical difficulties, this implementation of the calculus was not developed any further.

4 Multi-Tape Operations

A number of operations which we have used in the compilation of rewrite rules and lexica into automata are defined below. These are all supported by the Prolog implementation.

When an operator $\texttt{Op}$ takes a number of arguments $(a_1, \cdots, a_k)$, the arguments are shown as a subscript, e.g. $\texttt{Op}_{(a_1,\cdots,a_k)}$; the parentheses are ignored if there is only one argument. When the operator is mentioned without reference to arguments, it appears on its own, e.g. $\texttt{Op}$.

Unless otherwise specified, operations which are defined on tuples of strings can be extended to sets of tuples and relations. For example, if S is a tuple of strings and $\texttt{Op}(S)$ is an operator defined on S, the operator can be extended to a relation R in the following manner

$$\texttt{Op}(R) = \{\, \texttt{Op}(S) \mid S \in R \,\}$$

Identity The identity operator is a generalisation of the $\texttt{Id}$ operation in (Kaplan and Kay, 1994).

Definition 1. Let L be a regular language. $\texttt{Id}_n(L) = \{\, X \mid X$ is an n-tuple of the form $\langle x, \cdots, x\rangle, x \in L \,\}$ is the n-way identity of L.

Construction 4.1 Let L be a regular language and let $M = (Q, \Sigma, \delta, q_0, F)$ be a 1-tape FSA that accepts L. We can construct an n-tape FSA which accepts $\texttt{Id}_n(L)$, $M' = (Q, \Sigma, \delta', q_0, F)$, where

$$\delta'(q, \langle a, \cdots, a\rangle) = \delta(q, a) \text{ for all } q \in Q \text{ and } a \in \Sigma$$

and $\langle a, \cdots, a\rangle$ is an n-tuple.

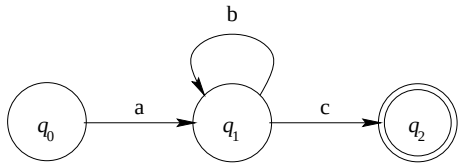
Example 1. Let $\Sigma = \{\, a, b, c \,\}$ and L be a regular language described by the regular expression ab^*c. $\texttt{Id}_3(L)$ is a 3-relation described by the regular expression *a:a:a b:b:b* c:c:c*. The automata for both expressions are given in Fig. 2.

Insertion and Removal The following $\texttt{Insert}$ operator is similar to the subscript operator in (Kaplan and Kay, 1994) in that it inserts symbols freely throughout a relation.

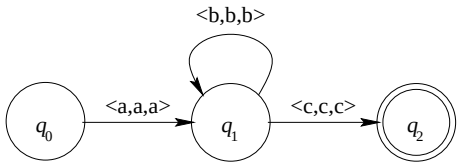
Definition 2. Let R be a regular relation over the alphabet Σ and let m be a set of symbols not necessarily in Σ. $\texttt{Insert}_m(R)$ inserts the relation $\texttt{Id}_n(a)$ for all $a \in m$, freely throughout R.

Remark. $\texttt{Insert}_m^{-1}$ (with $\texttt{Insert}_m^{-1} \circ \texttt{Insert}_m(R) = R$) removes all such instances if and only if m is disjoint from Σ.

Fig. 2. Id Operator



(a) ab^*c



(b) *a:a:a b:b:b* c:c:c*

Construction 4.2 Let $M = (Q, \Sigma, \delta, q_0, F)$ be an n-tape FSA which accepts the relation R, and m be a set of symbols; further, let $I_a = \mathtt{Id}_n(a)$ for all $a \in m$. We can construct an n-tape FSA which accepts $\mathtt{Insert}_m(R)$, $M' = (Q, \Sigma \cup m, \delta', q_0, F)$, where

$$\text{i) } \delta'(q, a) = \delta(q, a) \text{ for all } q \in Q \text{ and } a \in \Sigma^n$$
$$\text{ii) } \delta'(q, I_a) = q \text{ for all } q \in Q \text{ and } a \in m$$

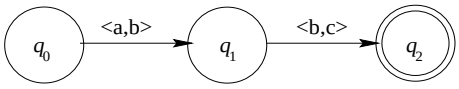
Conversely, one can construct an FSA for $\mathtt{Insert}^{-1}$.

Example 2. Let R be a 2-relation described by the regular expression *a:b b:c*. $R' = \mathtt{Insert}_{\{\alpha,\beta\}}(R)$ is a 2-relation described by the regular expression $(\alpha{:}\alpha \cup \beta{:}\beta)^*$ *a:b* $(\alpha{:}\alpha \cup \beta{:}\beta)^*$ b:c $(\alpha{:}\alpha \cup \beta{:}\beta)^*$. In a similar manner, $R = \mathtt{Insert}^{-1}_{\{\alpha,\beta\}}(R')$ if and only if $\{\alpha, \beta\}$ is disjoint from the alphabet of R. The automata for both expressions are given in Fig. 3.

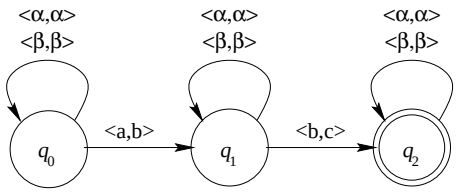
Substitution

Definition 3. Let S and S' be same-length n-tuples of strings over some alphabet Σ, $I = \mathtt{Id}_n(a)$ for some $a \in \Sigma$, and $S = S_1 I S_2 I \cdots S_k, k \geq 1$, such that S_i does not contain I – i.e. $S_i \in (\Sigma^n - \{I\})^*$. We say that $\mathtt{Substitute}_{(S',I)}(S) = S_1 S' S_2 S' \cdots S_k$ substitutes every occurrence of I in S with S'.

Fig. 3. Insert Operator



(a) $R = a{:}b\ b{:}c$



(b) $R' = \texttt{Insert}_{\{\alpha,\beta\}}(R)$

An algorithm for constructing an FSA for substitution is given in (Hopcroft and Ullman, 1979).

Example 3. Let R_1 be a 2-relation described by the regular expression $a{:}b\ \alpha{:}\alpha\ b{:}c$ and R_2 be a 2-relation described by the regular expression $x{:}x\ y{:}y$. $\texttt{Substitute}_{(R_2,\alpha:\alpha)}(R_1)$ is a 2-relation described by the regular expression $a{:}b\ x{:}x\ y{:}y\ b{:}c$. The automata for the above expressions are given in Fig. 4.

Cross Product The following operation takes two-same-length regular relations and returns their cross product.

Definition 4. Let R_1 be a same-length regular m-relation and R_2 be a same-length regular n-relation. $R_1 \times R_2$ is the same-length regular $(m{+}n)$-relation

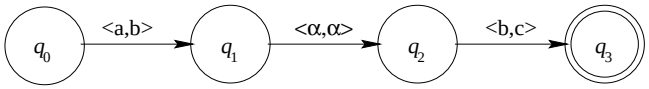
$$R = \{\, \langle a_1, \cdots, a_m, b_1, \cdots, b_n \rangle \mid \langle a_1, \cdots, a_m \rangle \in R_1, \langle b_1, \cdots, b_n \rangle \in R_2, \\ |a_1| = |b_1| \,\}$$

Remark. The last condition above, viz. $|a_1| = |b_1|$, ensures that the outcome is also a same-length relation.

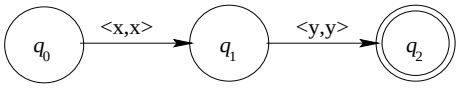
Construction 4.3 Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ be an m-tape FSA which accepts a same-length relation, and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ be an n-tape FSA which accepts a same-length relation. We can construct an $(m{+}n)$-tape FSA which accepts $M_1 \times M_2$,

$$M = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, [q_1, q_2], F_1 \times F_2)$$

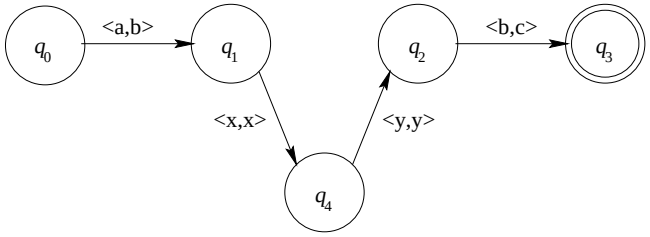
Fig. 4. `Substitute` Operator



(a) $R_1 = a{:}b\ \alpha{:}\alpha\ b{:}c$



(b) $R_2 = x{:}x\ y{:}y$



(c) $\texttt{Substitute}_{(R_2, \alpha:\alpha)}(R_1)$

where for all $p_1 \in Q_1, p_2 \in Q_2, \langle a_1, \ldots, a_m \rangle \in (\Sigma_1)^m$, and $\langle b_1, \ldots, b_n \rangle \in (\Sigma_2)^n$,

$$\delta([p_1, p_2], a_1 : \cdots : a_m : b_1 : \cdots : b_n) = [\delta_1(p_1, a_1 : \cdots : a_m), \delta_2(p_2, b_1 : \cdots : b_n)]$$

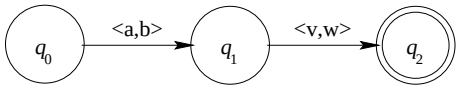
Example 4. Let R_1 be a same-length 2-relation described by the regular expression $a{:}b\ v{:}w$, and let R_2 be a same-length 3-relation described by the regular expression $c{:}d{:}e\ x{:}y{:}z$. $R_1 \times R_2$ is a 5-relation described by the regular expression $a{:}b{:}c{:}d{:}e\ v{:}w{:}x{:}y{:}z$. The automata for the above expressions are given in Fig. 5.

Projection

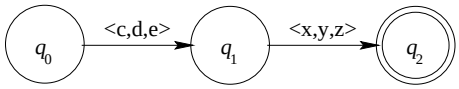
Definition 5. Let $S = \langle s_1, \ldots, s_n \rangle$ be a tuple of strings. The operator $\texttt{Project}_i(S)$, for some $i \in \{1, \ldots, n\}$, denotes the tuple element s_i. $\texttt{Project}_i^{-1}(S)$, for some $i \in \{1, \ldots, n\}$, denotes the $(n-1)$-tuple $\langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n \rangle$.

Example 5. Let $S = \langle$a,b,c$\rangle$. Then $\texttt{Project}_1(S) =$ a, $\texttt{Project}_2(S) =$ b and $\texttt{Project}_3^{-1}(S) = \langle$a,b$\rangle$.

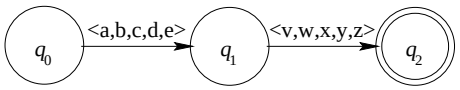
Fig. 5. × (Cross Product) Operator



(a) $R_1 = a{:}b\ v{:}w$



(b) $R_2 = c{:}d{:}e\ x{:}y{:}z$



(c) $R_1 \times R_2$

Constructing an FSA for `Project` and $\texttt{Project}^{-1}$ can be achieved by modifying the labels on transitions in a straightforward manner.

Summary of Operators The above operators are given in Tabble 1. In addition to the above operators, the following discussion makes use of '+' (or ∪) for disjunction and '−' for subtraction.

Table 1. N-way Operators

Operator	Description	Domain Type	Range Type
×	cross product	same-length relation	same-length relation
`Id`	n-way identity	language	relation
`Insert`	insertion	relation	relation
$\texttt{Insert}^{-1}$	removal	relation	relation
`Project`	projection	tuple of strings	string
$\texttt{Project}^{-1}$	inverse projection	tuple of strings	tuple of strings / string
`Substitute`	Substitution	relation	relation

5 An Evaluation

Table 2 shows some approximate execution times in seconds for minimising a particular class of automaton using FIRE Lite (v1.1, May 1994) (Watson, 1996), Grail (2.5, March 1996) (Raymond and Wood, 1996) and two versions of the finite-state calculus in Prolog: `fsa` supports arbitrary Prolog terms as transition labels while `fsq` uses only atoms and is the fastest version. FIRE Lite and Grail are both written in C++ while `fsa` and `fsq` were running with SICStus Prolog. In each case the machine used was a DECstation 5000 Model 133 running ULTRIX. Both FIRE Lite and Grail offer several minimisation functions; the functions that apply a version of Hopcroft's algorithm were chosen for testing as this is the algorithm used by `fsa` and `fsq`.

The automaton used was in each case the automaton with $n + 1$ states that recognises the string a^n. This highly regular automaton may be a worst or a best case for some of the programs. However, in our experience of using `fsq` for a variety of different and complex automata we found that the execution time for minimising an automaton was always about 10 ms per state, which suggests that the worst-case time complexity for that implementation really is about linear (or $n \log n$ as predicted) rather than about quadratic. The other implementations appear to run in quadratic time or worse, apparently because of an embedded linear search.

Table 2. Evalutaion

n	FIRE Lite	Grail	`fsa`	`fsq`
10	2			
20	24			
30	106			
40	388			
50		1	3.9	0.5
100		1	14.0	1.0
200		4	53.3	2.0
400		13	207.4	4.0
600		29		5.9
800		54		8.0

Note that these timing should not be taken as indicative of the overall performance of these software packages. Both Grail and FIRE Lite offer Brzozowski's minimisation algorithm as an alternative: although Brzozowski's algorithm is exponential in the worst case there are some indications that it is in practice more efficient than Hopcroft's algorithm when applied to the kind of automata used in practice. However, the observed quadratic behaviour of FIRE Lite's and Grail's implementations of Hopcroft's algorithm raises the worry that Hopcroft's

algorithm may not previously have been given a fair trial: the figures seem to confirm that Hopcroft's algorithm is difficult to implement correctly.

6 Illustration

This section illustrates the use of our finite-state calculus engine in implementing the compilation of regular rewrite rules into automata. We do so by showing how context-restriction two-level rules are compiled according to the algorithm provided by (Kaplan and Kay, 1994, §7.3).

The Prolog term `regexp_to_fsa(+RegExp,?Automaton)` constructs the FSA `Automaton` for the regular expression `RegExp`. The latter takes the following operators (in increasing precedence order): infix `^` for concatenation, infix `+` or `\/` (or simply a list) for union, infix `&` or `/\` for intersection, infix `-` or `\` for difference, postfix `*` for Kleene, postfix `+` for Kleene plus, prefix `~` or `-` for negation, and prefix `#` for reference to another automaton. The predicate `t(+terminal)` denotes a terminal symbol.

6.1 Kaplan and Kay's Algorithm

(Kaplan and Kay, 1994) define the following two regular relations. The 'if prefix then suffix' regular relation is expressed by:

$$\textit{If-P-then-S}(L_1, L_2) = \overline{L_1 \overline{L_2}}$$

Similarly, the 'if suffix then prefix' regular relation is expressed by:

$$\textit{If-S-then-P}(L_1, L_2) = \overline{\overline{L_1} L_2}$$

A context restriction two-level rule takes the form

$$\tau \Rightarrow \lambda \_\_ \rho$$

where τ, λ and ρ are subsets of Σ^* for some alphabet of tuples of symbols Σ. τ denotes the center of the rule, and λ and ρ denote left and right context pairs, respectively. For example, the rule

$$\text{n:m} \Rightarrow \_\_ \text{p:p}$$

states that 'n' rewrites as 'm' before a 'p'; hence, **inpossible* becomes *impossible*.

Such rules are expressed by the *Restrict* relation,

$$\textit{Restrict}(\tau, \lambda, \rho) = \textit{If-S-then-P}(\Sigma^* \lambda, \tau \Sigma^*) \cap \textit{If-P-then-S}(\Sigma^* \tau, \rho \Sigma^*)$$

The first component of *Restrict* states: it is *not* the case that τ is *not* preceded by λ; in other words, if τ occurs, it must be preceded by λ. Similarly, the second component states: it is *not* the case that τ is *not* followed by ρ. In both components, Σ^* on both sides allows any number tuples of symbols from the alphabet to precede and follow the description. The intersection forces both conditions.[5]

[5] (Kaplan and Kay, 1994) go on to describe a variation of this description in order to

6.2 Implementation

The finite-state calculus engine can be used without the user having to know about the details of implementing the automata. In fact, the user may chose from a number of implementations of the calculus as indicated earlier.

The 'if prefix then suffix' and 'if suffix then prefix' relations can be simply made into Prolog terms as follows:

```
if-p-then-s(L1, L2, FSA):-
   regexp_to_fsa(~(L1^(~L2)), FSA).

if-s-then-p(L1, L2, FSA):-
   regexp_to_fsa(~(~L1^L2), FSA).
```

These in turn can be used by another Prolog term which implements the *Restrict* operator as follows:

```
restrict(Sigma, T, L, R, FSA):-
   if-s-then-p(Sigma^L, T^Sigma, FSA1),
   if-p-then-s(Sigma^T, R^Sigma, FSA2),
   regexp_to_fsa((#FSA1)/\(#FSA2), FSA).
```

Here `T`, `L` and `R` represent τ, λ and ρ, respectively.

7 Conclusion

This paper described our Prolog implementation of a multi-tape finite-state engine and its use in solving natural language and speech problems. It also demonstrated the advantages of having an algebraic interface to the finite-state calculus, most importantly ease of implementing algorithms. The paper also showed that it is possible to make a practicable implementation of a finite-state calculus system in Prolog.

References

Carlsson, M. 1996. Personal communication to Edmund Grimley-Evans and posting to the Usenet news groups `comp.lang.prolog` (16 july 1996).

Eilenberg, S. 1974. *Automata, Languages, and Machines*, volume A. Academic Press.

Elgot, C. and J. Mezei. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9:47–68.

allow for the possibility that the context substring of one application might overlap with the center and context portions of a preceding one. For our illustrative purposes, however, we need not go into such details.

Hopcroft, J. and J. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

Kaplan, R. and M. Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–78.

Kay, M. 1987. Nonconcatenative finite-state morphology. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 2–10.

Kiraz, G. 1994. Multi-tape two-level morphology: a case study in Semitic nonlinear morphology. In *COLING-94: Papers Presented to the 15th International Conference on Computational Linguistics*, volume 1, pages 180–6.

Kiraz, G. [Forthcoming]. *Computational Approach to Nonlinear Morphology: with emphasis on Semitic languages.* Cambridge University Press.

Mealy, G. 1955. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–79.

Nivat, M. 1968. Transductions des languages de Chomsky. *Ann. Inst. Fourier (Grenoble)*, 18:339–455.

O'Keefe, R. 1990. *The Craft of Prolog.* The MIT Press.

Rabin, M. and D. Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–25. Reprinted in Moore, E. (ed.) *Sequential Machines.* Addison-Wesley, Reading, Massachussetts, 1964, pp. 63-91.

Raymond, D. and D. Wood. 1996. The Grail papers. Technical report, University of Western Ontario, Department of Computer Science. Technical Report TR-491.

Roche, E., editor. 1997. *Finite-State Language Processing.* MIT Press.

Swedish Institute of Computer Science, 1995. *SICStus Prolog User's Manual.*

Watson, B. 1995. *Taxonomies and Toolkits of Regular Language Algorithms.* Ph.D. thesis, Technische Universiteit Eindhoven.

Watson, B. 1996. Implementing and using finite automata toolkits. In *ECAI Workshop on Extended Finite-State Models of Language.*

This article was processed using the LaTeX macro package with LLNCS style