A Bimachine Compiler for Ranked Tagging Rules

Wojciech Skut, Stefan Ulrich and Kathrine Hammervold
Rhetorical Systems
4 Crichton's Close, Edinburgh
EH8 8DT
Scotland
wojciech@rhetorical.com

Abstract

This paper describes a novel method of compiling ranked tagging rules into a deterministic finite-state device called a *bimachine*. The rules are formulated in the framework of *regular rewrite operations* and allow unrestricted regular expressions in both left and right rule contexts. The compiler is illustrated by an application within a speech synthesis system.

1 Motivation

In rule-based tagging, linguistic objects (e.g. phonemes, syllables, or words) are assigned linguistically meaningful labels based on the context. Each instance of label assignment is licensed by a *tagging rule* typically specifying that label ψ can be assigned to item ϕ if ϕ is preceded by a pattern λ and followed by a pattern ρ. The patterns λ and ρ are usually formulated as regular expressions over the input alphabet, but may also range over output labels.

The nature of the tagging task suggests a formalisation in terms of *finite-state transducers* (FSTs). More precisely, the task can be viewed as an instance of string rewriting. In this framework, a tagging rule is interpreted as a *regular rewrite rule* $\phi \rightarrow \psi / \lambda \_ \rho$.

Several methods have been proposed for the compilation of such rules into FSTs (Kaplan and Kay, 1994; Mohri and Sproat, 1996; Gerdemann and van Noord, 1999). A rewrite rule is converted into a number of transducers, which are combined by means of transducer composition, yielding an FST that implements the actual rewrite operation.

Typically, tagging is carried out by a set of rules $R_i : \phi_i \rightarrow \psi_i / \lambda_i \_ \rho_i$, $i = 1 \ldots n$, which may overlap and/or conflict. A regular rule compiler should not just convert the rules into separate transducers $T_1 \ldots T_n$. For efficiency reasons, it is highly desirable to convert them into a single machine in a way that determines how rule conflicts should be resolved.

There are two basic options as to how the rule transducers can be combined.

- The rules can be associated with numerical *costs*, which translate as transition weights in the compilation step. In this formalisation, the union $T_1 \cup \ldots \cup T_n$ is a *weighted finite-state transducer* (WFST). This transducer is typically non-deterministic, but the weights make it possible to find the optimal path efficiently (as an instance of the single-source shortest paths problem).
- The rules can be explicitly ranked. In such a case, *priority union* (Karttunen, 1998), or an equivalent operator, can be used to combine $T_1, \ldots, T_n$ into a single unambiguous FST which is then turned into a deterministic device (Skut et al., 2004).

The work reported here pursues the latter strategy. Although less powerful and flexible than e.g. probabilistic approaches, it has the advantage of efficiency: once the rules have been compiled, rewriting an input sequence of t symbols boils down to t lookups in a transition table ($2t$ in case of a bimachine, see below).[1]

Several compilation methods have been proposed for creating a deterministic machine out of a set of rules (Laporte, 1997; Roche and Schabes, 1995; Hetherington, 2001). However, most of them impose strong restrictions on the form of contextual constraints: λ and ρ are restricted to single symbols (Hetherington, 2001), or acyclic regular expressions (Laporte, 1997).

Skut et al. (2004) describe a more powerful rewrite rule compiler that does not impose such constraints on ϕ, λ and ρ. Each rule R_i is compiled into an unambiguous FST T_i that inserts a marker $\#_i$ at the beginning of every match of

[1] With hand-written rules, the simple ranking is actually an advantage since a more complex rule interaction typically affects the transparency of the system.

ϕ_i preceded by an instance of λ_i and followed by an instance of ρ_i. While ρ_i and λ_i may contain markers inserted by other rules, ϕ_i must be marker-free. The composition $T_1 \circ \ldots \circ T_n$ of the rule transducers yields an FST that inserts rule markers into the input string, resolving rule conflicts according to the explicit ranking of rules. Composed with a transducer that performs the actual rewrite operation, it produces an unambiguous FST which implements the required combination of rules.

Two problems arise with this approach.

- Although the resulting FST is unambiguous (i.e., implements a function), it may be non-determinisable (Mohri, 1997a; Laporte, 1997).
- The composition operation used to combine the ranked rule transducers quickly creates large non-deterministic FSTs, resulting in slow compilation and high memory requirements.

The remedy to the first problem is straightforward: since the resulting FST implements a function, it can be compiled into a bimachine, i.e. an aggregate of a left-to-right and a right-to-left deterministic finite-state automaton (FSA) associated with an output function. The application of such a bimachine to a string involves running both automata (in the respective directions) and determining the symbols emitted by the output function (cf. section 2.1).

The simplest option is thus first to create the rule transducers, then compose them into a (non-deterministic) FST, and finally apply a bimachine construction method (1996). However, such a solution will not eliminate the inefficiency caused by expensive rule composition.

Thus, we have developed a compilation method that constructs the left-to-right and the right-to-left automaton of the resulting bimachine directly from the patterns without having to construct and then to compose the rule transducers. The efficiency of the compiler is increased by employing *finite-state automata* instead of FSTs, since algorithms used to process FSAs are typically faster than the corresponding transducer algorithms. Furthermore, the resulting (intermediate) structures are significantly smaller than in the case of FSTs. This leads to much faster compilation and smaller finite-state machines.

2 Formalization

2.1 Definitions and Notation

In the following definitions, Σ denotes a finite input alphabet. Δ is a finite output alphabet.

A *deterministic finite-state automaton* (DFSA) is a quintuple $A = (\Sigma, Q, q_0, \delta, F)$ such that:

Q is a finite set of states;

$q_0 \in Q$ is the initial state of A;

$\delta : Q \times \Sigma \to Q$ is the transition function of A;

$F \subset Q$ is a non-empty set of final states.

A *sequential transducer* (ST) is defined as a 7-tuple $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ such that $(\Sigma, Q, q_0, \delta, F)$ is a DFSA, and $\sigma(q, a)$ is the output associated with the transition from state q via symbol a to state $\delta(q, a)$.

The functions δ and σ can be extended to the domain $Q \times \Sigma^*$ by the recursive definition: $\delta^*(q, \epsilon) = q$, $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$, $w \in \Sigma^*$, $a \in \Sigma$ and $\sigma^*(q, \epsilon) = \epsilon$, $\sigma^*(q, wa) = \sigma^*(q, w)\sigma(\delta^*(q, w), a)$.

A *bimachine* B is a triple $(\overrightarrow{A}, \overleftarrow{A}, h)$ such that:

$\overrightarrow{A} = (\Sigma, \overrightarrow{Q}, \overrightarrow{q}_0, \overrightarrow{\delta})$ is a left-to-right FSA (there is no concept of final states in a bimachine);

$\overleftarrow{A} = (\Sigma, \overleftarrow{Q}, \overleftarrow{q}_0, \overleftarrow{\delta})$ is a right-to-left FSA;

$h : \overrightarrow{Q} \times \Sigma \times \overleftarrow{Q} \to \Delta^*$ is the *output function.*

Applied to a string $u = a_1 \ldots a_t$, B produces a string $v = b_1 \cdot \ldots \cdot b_t$, such that $b_i \in \Delta^*$ is defined as follows:

$$b_i = h(\overrightarrow{\delta}(\overrightarrow{q}_0, a_1 \ldots a_{i-1}), a_i, \overleftarrow{\delta}(\overleftarrow{q}_0, a_t \ldots a_{i+1}))$$

An unambiguous finite-state transducer can always be converted into a bimachine (Berstel, 1979; Roche and Schabes, 1996). This property makes bimachines an attractive tool for deterministic processing, especially since not all unambiguous transducers are determinisable (sequentiable).

2.2 Rules, Rule Ordering and Priorities

As mentioned in section 1, the tagging rule formalism is formulated in the framework of regular rewrite rules (Kaplan and Kay, 1994). Input to the rule compiler consists of a set of rules.

$$\begin{aligned} R_1 : \phi_1 &\rightarrow \psi_1 / \lambda_1 \_ \rho_1 \\ &\vdots \\ R_n : \phi_n &\rightarrow \psi_n / \lambda_n \_ \rho_n \end{aligned}$$

A rule $\phi \rightarrow \psi / \lambda \_ \rho$ states that label ψ is assigned to object ϕ (called the *focus* of the rule) if ϕ is preceded by a left context λ and followed by a right context ρ. The context descriptions λ, ρ and ϕ are formulated as regular expressions over the input alphabet Σ.

The rules may conflict, in which case the ambiguity is resolved based on the order of the rules in the grammar. If a rule R_i fires for an object s_k in the input sequence $s_1 \dots s_t$, it blocks the application of all rules R_j, $j > i$, to s_k.

Thus the operational semantics of the rules may be stated as follows: A rule R_i fires if:

(a) no other rule R_j, $j < i$, is applicable in the same context;

(b) the substring $s_1 \dots s_{k-1}$ matches the regular expression $\Sigma^* \lambda_i$;

(c) the substring $s_k \dots s_t$ matches the regular expression $\phi_i \rho_i \Sigma^*$.

This basic formalism imposes two conditions on the rules:

- λ and ρ are regular expressions over Σ;
- $\phi \in \Sigma$ is a single object.

These two assumptions restrict the expressive power of the formalism compared to general regular rewrite rules (Kaplan and Kay, 1994; Mohri and Sproat, 1996) in that they do not allow output symbols on either side of the context and only admit rule foci of length one. Although these restrictions are essential for the initial basic formalism, we show in section 3 how to extend it so that the compiler can accept rules with both input and output symbols in the left context. As for the length of the focus, it is important to bear in mind that the formalism is primarily intended for tagging rules which usually do not cover longer foci.

2.3 Matching of Context Patterns

The basic idea in the new compilation method is to convert the patterns λ_i and $\rho_i \phi_i$, $i = 1, \dots n$, directly into the left-to-right and right-to-left acceptor of a bimachine without having to perform the fairly expensive operations required by the transducer-based approaches.

Key to the solution is the function $SimultMatch_{\beta_1 \dots \beta_n} : \Sigma^t \rightarrow (2^{\mathbf{N}})^{t+1}$, $t \in \mathbf{N}$, which, given a collection $\{\beta_1, \dots, \beta_n\}$ of regular expressions, maps a sequence of symbols $s_1 \dots s_t$ to a sequence of $t+1$ sets of indices corresponding to the matching patterns at the respective position (position 0 corresponds to the beginning of the string, I denotes the set $\{1, \dots, n\}$ of rule indices):

$$SimultMatch_{\beta_1 \dots \beta_n}(s_1 \dots s_t)[k] = \{j \in I : \Sigma^* \beta_j \text{ matches } s_1 \dots s_k\}$$

This construct can be implemented as a pair $(A_{\beta_1 \dots \beta_n}, \tau)$ such that:

$A_{\beta_1 \dots \beta_n} = (\Sigma, Q, q_0, \delta, F)$ is a finite-state automaton that encodes in its states (Q) information about the matching patterns.

$\tau : Q \rightarrow 2^I$ is a function mapping the states of A to sets of indices corresponding to matching regular expressions: $\tau(\delta^*(q_0, w)) = \{j \in I : \Sigma^* \beta_j \text{ matches } w\}$.

In order to construct $SimultMatch_{\beta_1 \dots \beta_n}$, we introduce a marker symbol $\$_j \notin \Sigma$ for each β_j. Let $\Sigma_\$ = \Sigma \cup \bigcup_{j \in I} \{\$_j\}$ be the extended alphabet. Let $\tilde{A} = (\Sigma_\$, \tilde{Q}, \tilde{q_0}, \tilde{\delta}, \tilde{F})$ be a deterministic finite acceptor for the regular expressions $\Sigma^* \bigcup_{j \in I} \beta_j \$_j$.

An important property of the automaton $\tilde{A}$ is that $w \in \Sigma^*$ is an instance of a pattern $\Sigma^* \beta_j$ if and only if $q = \tilde{\delta}^*(\tilde{q_0}, w)$ is defined and there exists a transition from q by $\$_j$ to a final state: $\tilde{\delta}(\tilde{\delta}^*(\tilde{q_0}, w), \$_j) \in F$. Now we can define the function $\tau : Q \rightarrow 2^I$:

$$\tau(q) = \{j \in I : (q, \$_j) \in Dom(\tilde{\delta}) \wedge \tilde{\delta}(q, \$_j) \in F\}$$

Obviously, if $\tilde{A}$ enters state q after consuming a string $w \in \Sigma^*$, $\tau(q)$ is the set of all indices j such that w matches $\Sigma^* \beta_j$.

The automaton $A_{\beta_1 \dots \beta_n} = (\Sigma, Q, q_0, \delta, Q)$ can now be constructed from $\tilde{A}$ by restricting it to the alphabet Σ (which includes trimming away the unreachable states) and making all its states final so that it accepts all strings $w \in \Sigma^*$:

$$Q = \{q \in \tilde{Q} : \exists w \in \Sigma^* \quad \tilde{\delta}^*(\tilde{q_0}, w) = q\}$$
$$q_0 = \tilde{q_0}$$
$$\delta = \tilde{\delta}|_{Q \times \Sigma \times Q}.$$

The resulting construct $(A_{\beta_1 \dots \beta_n}, \tau)$ makes it possible to simultaneously match a collection of regular expressions.

2.4 Bimachine Compilation

Using the construct *SimultMatch*, we can determine all the matching left and right contexts at any position k in a string $w = a_1 \dots a_t$.

The value of $SimultMatch_{\lambda_1 \ldots \lambda_n}(w)[k-1]$ is the set of all rule indices i such that λ_i matches the string $a_1 \ldots a_{k-1}$. $SimultMatch_{(\phi_1\rho_1)^{-1}\ldots(\phi_n\rho_n)^{-1}}(w^{-1})[t-k]$ is the set of all rule indices i such that $\phi_i\rho_i$ matches the remainder $s_k \ldots s_t$ of w. Obviously, the intersection $SimultMatch_{\lambda_1 \ldots \lambda_n}(w)[k-1] \cap SimultMatch_{(\phi_1\rho_1)^{-1}\ldots(\phi_n\rho_n)^{-1}}(w^{-1})[t-k]$ is exactly the set of all matching rules at position k. The minimal element of this set is the index of the firing rule.[2]

Now if $(\overrightarrow{A}, \overrightarrow{\tau}) := SimultMatch_{\lambda_1 \ldots \lambda_n}$, and $(\overleftarrow{A}, \overleftarrow{\tau}) := SimultMatch_{(\phi_1\rho_1)^{-1}\ldots(\phi_n\rho_n)^{-1}}$, then the tagging task is performed by the bimachine $B = (\overrightarrow{A}, \overleftarrow{A}, h)$, where the output function $h : \overrightarrow{Q} \times \Sigma \times \overleftarrow{Q} \to \Delta^*$ is defined as follows:

$$h(\overrightarrow{q}, a, \overleftarrow{q}) \quad = \quad \psi_{\min(\overrightarrow{\tau}(\overrightarrow{q}) \cap \overleftarrow{\tau}(\overleftarrow{\delta}(\overleftarrow{q}, a)))}$$

The output function can be either precompiled (e.g., into a hash table), or – if the resulting table is too large – the intersection operation can be performed at runtime, e.g. using a bitset encoding of sets.[3]

3 Extensions

The compiler introduced in the previous section can be extended to handle more sophisticated rules and search/control strategies.

3.1 Output Symbols in Left Contexts

The rule formalism can be extended by including output symbols in the left context of a rule. This extra bit is added in the form of a regular expression π ranging over the output symbols, which can be represented by rule IDs $r_k \in I$. The rules then look as follows:

$$R_i = \phi_i \quad \to \quad \psi_i/\pi_i : \lambda_i \_ \rho_i$$

Such a rule fires at a position k in string $s_1 \ldots s_t$ if an extra condition (d) holds in addition to the conditions (a)–(c), formulated in section 2.2:

(d) The IDs $r_1 \ldots r_{k-1}$ of the firing rules match $I^*\pi_i$.

In order to enforce condition (d), we use the *SimultMatch* construct introduced in section 2.3. For that, the patterns $\pi = \{\pi_1, \ldots, \pi_n\}$ are compiled into an instance of $SimultMatch_\pi = (A_\pi, \tau_\pi)$. A_π is an FSA, so $A_\pi = (I, Q_\pi, q_0^\pi, \delta_\pi)$. It follows from the construction of $SimultMatch_\pi$ that the function $\tau_\pi : Q_\pi \to 2^I$ has the following property:

$$\tau_\pi(\delta_\pi^*(q_0^\pi, r_1 \ldots r_k)) = \{j \in I : r_1 \ldots r_k \text{ matches } \pi_j\}$$

In other words, an action ψ_{r_k} is admissible at position k if $r_k \in \tau_\pi(\delta_\pi^*(q_0, r_1 \ldots r_{k-1}))$. Thus, the tagging task (according to the extended strategy (a)–(d)) is performed by the formal machine $M = (\overrightarrow{A}, A_\pi, \overleftarrow{A}, h)$, where $\overrightarrow{A}$ and $\overleftarrow{A}$ are as in section 2.4, and $h : \overrightarrow{Q} \times Q_\pi \times \Sigma \times \overleftarrow{Q} \to \Delta^*$ is defined as follows:[4]

$$h(q, q^\pi, a, q') = \psi_{r_k},$$

where

$$r_k := \min(\overrightarrow{\tau}(q) \cap \tau_\pi(q^\pi) \cap \overleftarrow{\tau}(\overleftarrow{\delta}(q', a)))$$

In this formula, $\overrightarrow{q}$ and $\overleftarrow{q}$ are as in the basic bimachine introduced in section 2.4. q^π is the state of A_π after consuming the rule IDs $r_1 \ldots r_{k-1}$: $q^\pi := \delta_\pi^*(q_0^\pi, r_1 \ldots r_{k-1})$.

In order to determine the tagging actions for an input sequence $w = a_1 \ldots a_t$, the automaton $\overleftarrow{A}$ is first run on w^{-1}. Then both $\overrightarrow{A}$ and A_π are run on w in parallel. In each step k, the states $\overrightarrow{\delta}(\overrightarrow{q}_0, a_1 \ldots a_{k-1})$, $\overleftarrow{\delta}(\overleftarrow{q}_0, a_t \ldots a_k)$ as well as the sequence $r_1 \ldots r_{k-1}$ of already executed actions are known, so that the ψ_{r_k}'s can be determined incrementally from left to right.

3.2 Alternative Control Strategies

Our rule compilation method is very flexible with respect to control strategies. By intersecting the sets of rule IDs $\overrightarrow{\tau}(\overrightarrow{\delta}(\overrightarrow{q}_0, a_1 \ldots a_{k-1}))$

[2] In order to ensure that the above formula is always valid, we assume that the rule with the highest index (R_n) matches all left and right contexts (i.e., $\lambda_n = \rho_n = \Sigma^*, \phi_n = \bigcup_{\sigma \in \Sigma}\{\sigma\}$), and ψ_n is a vacuous action. If none of the other rules fire, the formalism defaults to R_n.

[3] In the actual implementation of the tagger, h has been replaced by a function $g : \overrightarrow{Q} \times \overleftarrow{Q} \to \Delta^*$ defined as:

$$g(\overrightarrow{q}, \overleftarrow{q}) \quad = \quad \psi_{\min(\overrightarrow{\tau}(\overrightarrow{q}) \cap \overleftarrow{\tau}(\overleftarrow{q}))}$$

The translation of the k-th symbol in a string $w = a_1 \ldots a_t$ is then determined by the formula

$$g(\overrightarrow{\delta}(\overrightarrow{q}_0, a_1 \ldots a_{k-1}), \overleftarrow{\delta}(\overleftarrow{q}_0, a_t \ldots a_k))$$

which is easier to compute.

[4] In order to make sure the definition of h is always valid, we assume that the rule with the highest index (n) matches all possible contexts (i.e., $\pi_n = I^*, \lambda_n = \rho_n = \Sigma^*$ and $\phi_n = \bigcup_{\sigma \in \Sigma}\{\sigma\}$).

and $\overleftarrow{\tau}(\overleftarrow{\delta}(\overleftarrow{q}_0, a_t \dots a_k))$, $1 \leq k \leq t$, one can determine the set of all matching rules for each position in the input string. In the formalism presented in section 2.4, only one rule is selected, namely the one with the minimal ID. This is probably the most common way of handling rule conflicts, but the formalism does not exclude other control strategies.

Simultaneous matching of all rules: This strategy is particularly useful in the machine-learning scenario, e.g. in computing scores in transformation-based learning (Brill, 1995). Note that the simple context rules used by Brill (1995) may be mixed with more sophisticated hand-written heuristics formulated as regular expressions while still being subject to scoring. As shown in section 2.2, taggers using unrestricted regular context constraints are not sequentiable, and thus cannot be implemented using ST-based rule compilation methods (Roche and Schabes, 1995).

N-best/Viterbi search: Instead of a strict ranking, the rules may be associated with probabilities or scores such that the best sequence of actions is picked based on global, per-sequence, optimisation rather than on a sequence of greedy local decisions. In order to implement this, we can use the extended formalism introduced in section 3.1 with a slight modification: in each step, we keep N best-scoring paths rather than just the one determined by the selection of the locally optimal action ψ_{r_k} for $1 \leq k \leq t$.

4 An Application

In this section, we describe how our bimachine compiler has been applied to the task of homograph disambiguation in the rVoice speech synthesis system.

Each module in the system adds information to a structured relation graph (HRG), which represents the input sentence or *utterance* to be spoken (Taylor et al., 2001). The HRG consists of several *relations*, which are structures such as lists or trees over a set of *items*. The homograph tagger works on a *list* relation, where the items represent words. Each item has a feature structure associated with it, the most relevant features for our application being the *name* feature representing the normalised word and the *pos* feature representing the part-of-speech (POS) of the word.

The assignment of POS tags is done by a statistical tagger (a trigram HMM). Its output is often sufficient to disambiguate homographs, but in some cases POS cannot discriminate between two different pronunciations, as in the case of the word *lead*: *they took a 1-0 lead* vs. *a lead pipe* (both nouns). Furthermore, the statistical tagger turns out to be less reliable in certain contexts. The rule-based homograph tagger is a convenient way of fixing such problems.

The grammar of the homograph tagger consists of a set of ordered rules that define a mapping from an item to a *sense ID*, which uniquely identifies the phonetic transcription of the item in a pronunciation lexicon.

For better readability, we have changed the rule syntax. Instead of $\phi \rightarrow \psi / \lambda \_ \rho$, we write:

$$\lambda / \phi / \rho \rightarrow \psi$$

where λ, ϕ and ρ are regular expressions over feature structures. The feature structures are written $[f_1 = v_1 \dots f_k = v_k]$, where f_i is a feature name and v_i an atomic value or a disjunction of atomic values for that feature. Each attribute-value pair constitutes a separate input alphabet symbol. The alphabet also contains a special default symbol that denotes feature-value pairs not appearing in the rules. The symbol ψ stands for the action of setting the sense feature of the item to a particular sense ID.

The following are examples of some of the rules that disambiguate between the different senses of *suspects* (`sense=1` is the noun reading, `sense=2` the verb reading):

```
[name=that]
    / [name=suspects] /
                        -> [sense=2];

([pos=dt|cd]|[name=terror])
         / [name=suspects]/
                        -> [sense=1];
/ [name=suspects] /
              [name=that]
                        -> [sense=2];
/ [name=suspects] /     -> [sense=1];
```

Note that the last rule is a default one that sets *sense* to 1 for all instances of the word *suspects* where none of the other rules fire.

To explain the interaction of the rules, we will look at the following example:

$$the_1\ terror_2\ suspects_3\ that_4\ were_5\ in_6\ court_7$$

We can see that the second and the third rule match the context of word 3. The rule associated with the lower index fires, resulting in the value of *sense* being set to 1 on the item.

5 Performance Evaluation

To evaluate the performance of the new compilation method, we measured the compilation time and the size of the resulting structures for a set of homograph disambiguation rules in the format described in section 4. The results were compared to the results achieved using a compiler that converts each rule into an FST and then composes the FSTs and determinises the transducer created by composition (Skut et al., 2004). Both algorithms were implemented in C++ using the same library of FST/FSA classes, so the results solely reflect the difference between the algorithms.

The figures (1a)–(1c) on page 6 show the results of running both implementations on a Pentium 4 1.7 GHz processor for rule sets of different sizes. Figure (1a) shows the number of states, (1b) the number of transitions, and (1c) the compilation time. The numbers of states and transitions for A_2, the bimachine-based approach proposed in this paper, are the sums of the states and transitions, respectively, for the left-to-right and right-to-left acceptors. The left-to-right FSA typically has a much smaller number of states and transitions than the right-to-left FSA (only 10% of its states and 2–5% of its transitions) since it does not contain the regular expression for the rule focus.

While the figures show a substantial decrease in runtime for the bimachine construction method (A_2) compared to the FST-based approach A_1 (only 6.48 seconds instead of 115.29 seconds for the largest set of 40 rules in (1c)), the numbers of states and transitions are slightly larger for the bimachine. Typically the FSAs have about 25% more states and 35% more transitions than the corresponding STs in our test set. However, an FSA takes up less memory than an FST as there are no emissions associated with transitions and the output function h can be encoded in a very space-efficient way. As a result, the size of the compiled structure in RAM was down by almost 30% compared to the size of the original transducer.

6 Conclusion

The rule compiler described in this paper presents an attractive alternative to compila-

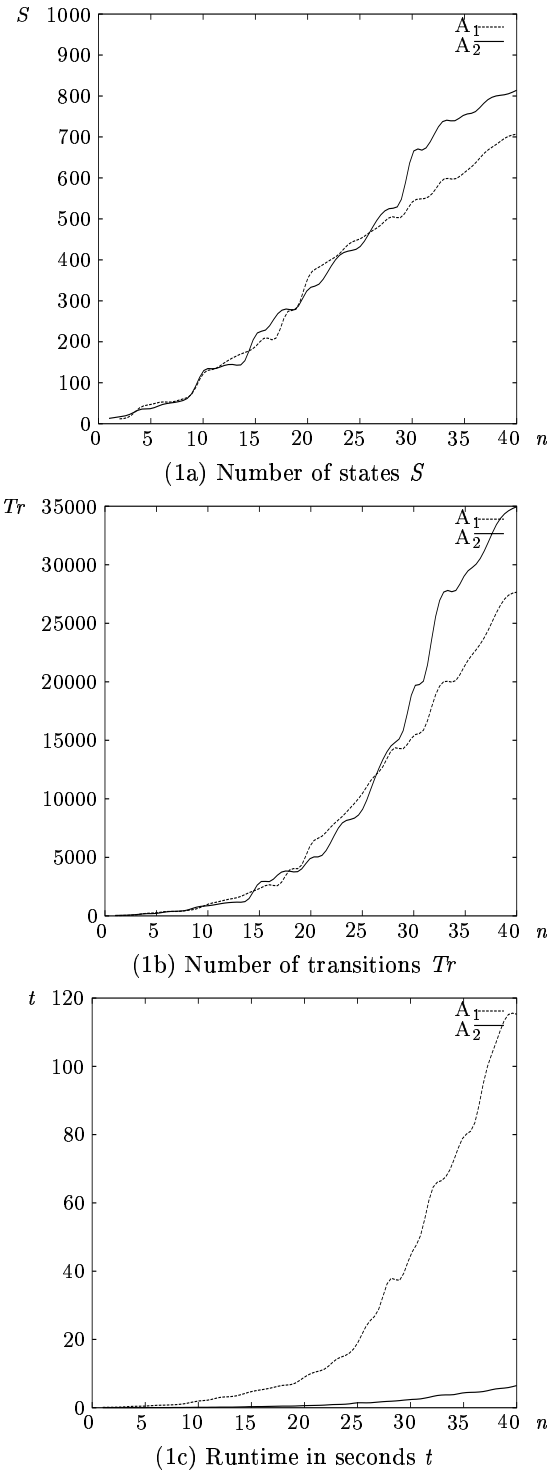

Figure 1: Comparison of two compilation algorithms. A_1 is an transducer-based construction method (Skut et al., 2004), A_2 the approach proposed in this paper. The figures show the numbers of states (1a), the number of transitions (1b) and the runtime (1c) depending on the size of the input rule file (n is the number of rules).

tion methods that use FST composition and complement in order to convert rewrite rules into finite-state transducers. The direct combination of context patterns into an acceptor with final outputs makes it possible to avoid the use of relatively costly FST algorithms. In the present implementation, the only potentially expensive routine is the creation of the deterministic acceptors for the context patterns. However, if the task is to create a deterministic device, determinisation (in its more expensive version for FSTs) is also required in the FST-based approaches (Skut et al., 2004). The experimental results presented in section 5 show that compilation speed is not a problem in practice. Should it become an issue, there is still room for optimisation. The potential bottleneck due to DFSA determinisation can be eliminated if we use a generalisation of the Aho-Corasick string matching algorithm (Aho and Corasick, 1975) in order to construct the deterministic acceptor for the language $\Sigma^* \bigcup_{j \in I} \beta_j \$_j$ while creating the *SimultMatch* construct (Mohri, 1997b).

By constructing a single deterministic device, we pursue a strategy similar to the compilation algorithms described by Laporte (1997) and Hetherington (2001). Our method shares some of their properties such as the restriction of the rule focus ϕ to one input symbol.[5] However, it is more powerful as it allows unrestricted (also cyclic) regular expressions in both the left and the right rule context. The practical significance of this extra feature is substantial: unlike phonological rewrite rules (the topic of both Laporte and Hetherington's work), homograph disambiguation does involve inspecting non-local contexts, which often pose a difficulty to the 3-gram HMM tagger used to assign POS tags in our system.

Although the use of our compiler is currently restricted to hand-written rules, the extensions sketched in section 3.2 make it possible to use it in a machine learning scenario (for both training and run-time application).

Our rule compiler has been applied successfully to a range of tasks in the domain of speech synthesis, including homograph resolution, post-lexical processing and phrase break prediction. In all these applications, it has proved to be a useful and reliable tool for the development of large rule systems.

[5] As pointed out in section 2.2, this restriction does not pose a problem as the compiler is primarily designed for rule-based tagging.

References

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: an aid to bibliographic search. In *Communications of the ACM*, 18(6):333–340.

Jean Berstel. 1979. *Transductions and Context-Free Languages.* Teubner Verlag.

Eric Brill. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

Dale Gerdemann and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Proceedings of EACL 99.*

I. Lee Hetherington. 2001. An efficient implementation of phonological rules using finite-state transducers. In *Proceedings of Eurospeech 2001.*

Ronald M. Kaplan and Martin Kay. 1994. Regular model of phonological rule systems. *Computational Linguistics*, pages 331–378.

Lauri Karttunen. 1998. The proper treatment of optimality in computational phonology. In Lauri Karttunen, editor, *FSMNLP'98: International Workshop on Finite State Methods in Natural Language Processing*, pages 1–12. ACL.

Eric Laporte. 1997. Rational transductions for phonetic conversion and phonology. In Emmanuel Roche and Yves Schabes, editors, *Finite-state language processing*, Language, Speech, and Communication Series, chapter 14, pages 407–428. MIT Press, Cambridge (Mass.).

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the Annual Meeting of the ACL*, pages 231–238.

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

Mehryar Mohri. 1997b. String-matching with automata. *Nordic Journal of Computing*, 4(2):217–231.

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

Emmanuel Roche and Yves Schabes. 1996. Introduction to finite-state devices in natural language processing. Technical report, Mitsubishi Electric Research Laboratories, TR-96-13.

Wojciech Skut, Stefan Ulrich, and Kathrine Hammervold. 2004. A flexible rule compiler for speech synthesis. In *Proceedings of Intelligent Information Systems 2004*, Zakopane, Poland.

Paul Taylor, Alan W. Black, and Richard Caley. 2001. Heterogeneous relation graphs as a formalism for representing linguistic information. *Speech Communication*, 33(1-2):153–174.