Preference-Driven Bimachine Compilation

An Application to TTS Text Normalisation

Wojciech Skut

Rhetorical Systems

Abstract

This paper describes a grammar formalism and a deterministic parser developed for text normalisation in the rVoice[1] text-to-speech (TTS) system. The rules are formulated using regular expressions and converted into a non-deterministic finite-state transducer (FST). At runtime, search is guided by parsing preferences which the user may associate with regular operators; the best solution is determined in a way similar to the directional evaluation of constraints in Optimality Theory. During compilation, the FST is converted into a bimachine, making deterministic parsing possible.

1 Motivation

Over the past decade, speech synthesis has become one of the most succesful commercial applications of natural language and speech processing technologies. During this time, it has established itself as a standard solution in call centre applications, telephone banking and several other areas. The reason for its success is rightly attributed to the emergence of high-quality unit selection synthesis methods (Hunt and Black 1996). However, as a result of this focus on *speech* processing, the *text* processing part of TTS typically receives less attention although a high quality text front end is indispensable for a good TTS system.

The task of the text front end in a TTS system is to convert raw input text into a sequence of unambiguous phonetic symbols. For example, the sentence *"On 22/5, Mr Brown had to pay an $80 fine"* may be transformed to the phonetic representation [on ðə twentɪ sekənd əv meɪ mɪstə braʊn hæd tə peɪ ən eɪtɪ dolə faɪn]. While mapping some of the words (e.g. *pay*) to phonetic symbols is straightforward, other parts of the input require a complex multi-stage transformation, e.g., *$80 → eighty dollar → [eɪtɪ dolə].* Non-trivial processing is required for numbers, dates (*22/5*), currency amounts (*$80*), abbreviations (*Mr*) and many other types of expressions. It is often context dependent (e.g. *$80 → eighty dollar/eighty dollars* as in *an $80 fine* vs. *he was fined $80*).

Typically, text processing is split into two main stages. In the first one, abbreviations, digits and symbols are rewritten as literal text (e.g. *On 22/5, Mr Brown had to pay an $80 fine → On the twenty-second of May, Mister Brown had to pay an eighty dollar fine*). Then, the actual phonetic rewrite takes place.

The term *text normalisation* commonly refers to the first processing stage, which is also the more difficult one. Due to the required broad coverage, text normalisation grammars tend to be very large and complex. As in other areas of NLP, disambiguation of alternative analyses is the main source of complexity: the string *1* may be expanded

[1] See `http://www.rhetorical.com/cgi-bin/demo.cgi`.

as *one*, *first*, *premier*, etc. depending on the context it appears in. Although it may be possible to learn such context-based disambiguation rules from data, existing text normalisation systems are typically rule-based, which is most probably due to two reasons.

Firstly, existing rule-based formalisms often come with very large and detailed grammars developed over years, an extremely valuable resource that might be wasted if one decided to abandon the rule-based paradigm. Secondly, text normalisation requires very high precision as we cannot afford, say, an account balance to be expanded incorrectly in an automated telephone banking application. The precision threshold is thus much higher than in those areas of NLP where data-driven approaches have become successful, e.g. in Machine Translation. This does not preclude a data-drived solution to the problem, but the quality restrictions imposed on the potential solutions make it very hard.

In effect, this paper focuses on disambiguation strategies for purely rule-based grammars. It shows how to implement a simple but intuitive and powerful disambiguation strategy within a system that is both efficient at runtime and expressive enough to handle typical text normalisation constructs. The use of the *finite-state transducer* (FST) framework provides for a good compromise between expressive power and computational tractability: a grammar is first compiled into a non-determinstic FST containing *markers* that express user-defined local parsing preferences. In an extra step, the preferences are used to turn the FST into a deterministic device called *bimachine*. The system scales well to large grammars and supports efficient and ergonomical grammar development, including interactive rule compilation and debugging.

2 Definitions and Notation

The following definitions are intended to clarify the notation for some basic finite-state constructs. The less well known notion of *bimachines* is introduced and explained in section 5.

Definition 1. *(***Nondeterministic FSA***) A* non-deterministic finite-state automaton *(NFSA) over input alphabet Σ is a quintuple $A = (\Sigma, Q, q_0, E, F)$, where Q is the set of states of A, $q_0 \in Q$ is the initial state, and $F \subset Q$ the set of final states, and $E \subset Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the set of A's transitions.*

Definition 2. *(***Deterministic FSA***) A* deterministic finite-state automaton *(DFSA) is a quintuple $A = (\Sigma, Q, q_0, \delta, F)$, where $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ the transition function, and F the set of final states. The symbol δ^* is used to denote the extension of the transition function to the domain $Q \times \Sigma^*$: $\delta^*(q, \epsilon) := q$, $\delta^*(q, ua) := \delta(\delta^*(q, u), a)$ for $a \in \Sigma, u \in \Sigma^*$.*

Definition 3. *(***Finite-State Transducer***) A* finite state transducer *(FST) $T = (\Sigma, \Delta^*, Q, q_0, E, F)$ over an input alphabet Σ and output alphabet Δ is defined identically to an NFSA except that each transition contains an output label, i.e., $E \subset Q \times (\Sigma \cup \{\epsilon\}) \times Q \times \Delta^*$. We will use the notation $e.source$, $e.input$, $e.output$ and $e.target$ to refer to the respective components of a transition $e \in E$.*

Definition 4. *(***Sequential Transducer***) A* sequential transducer *is the two-level counterpart of a DFSA. It is a 7-tuple* $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$*, such that* $(\Sigma, Q, q_0, \delta, F)$ *is a DFSA, and* $\sigma : Q \times \Sigma \rightarrow \Delta^*$ *is the* output function *defining the output of each transition.*

Definition 5. *(***Path, Reachability***) A path in an NFSA/FST is a sequence of transitions* $\pi = e_1, \ldots, e_t$ *such that* $e_j.target = e_{j+1}.source$ *for* $j = 1, \ldots, t-1$*. A path* π *consumes a string* w *if* $e_1.input \cdot \ldots \cdot e_t.input = w$*. In the following, the term path always denotes an* ϵ-cycle-free path*. In analogy to transitions, we will also write* $\pi.source$*,* $\pi.target$*, etc.*

The notation $q \stackrel{a:o}{\Rightarrow} q'$ *indicates that state* q' *is reachable from state* q *by exactly one transition consuming symbol* $a \in \Sigma$ *and emitting* o*, i.e.,* $(q, a, q', o) \in E$*. For a string* $u \in \Sigma^*$*,* $q \stackrel{u:o^*}{\Rightarrow} q'$ *denotes the reflexive-transitive closure of* $\Rightarrow$*, i.e., there is a path in* T *from* q *to* q' *consuming input* u *and emitting* $o \in \Delta^*$*.*

3 Grammars

Text normalisation consists of two sub-tasks: *parsing* (the identification of constructions that require normalisation) and *rewriting* (the actual string transformations, e.g. *$11 → eleven dollars*). Therefore, the formalism is split into *parsing rules* and *rewrite rules*.

The syntax of parsing rules is roughly based on common parser generators such as `yacc` or `bison`.[2] Each rule consists of a left-hand side non-terminal symbol (the *rule name*), a right-hand side specifying its *expansion* as a regular expression over terminal and non-terminal symbols, and an optional *rewrite statement* that calls *rewrite rules* in order to perform string rewriting, as shown below:

```
date
    ->
      day:$D [name="/"] month:$M
                      ([name="/"] year:$Y)?
 {"the" exp_ordinal($D) "of" $M exp_year($Y)};

day -> [name="(0?)[1-9]|[12][0-9]|3[01]"];

month -> [name="(0?)[1-9]|1[012]"];

year -> [name="[12][0-9][0-9][0-9]"];
```

Here, the left-hand side symbol `date` is expanded to `day` followed by the terminal `/`, the nonterminal `month`, and an optional instance of the nonterminal `year` preceded by the separator `/`. The symbols `day`, `month` and `year` expand to terminals according to the respective rules. Each terminal is a simple feature structure that denotes a token (in the above grammar fragment, only the feature `name` is used).

[2]Note, however, that the semantics of the rules, as discussed below in section 4.2, is different: while LR grammars such as the ones accepted by `yacc` and `bison` are expected to be unambiguous, our formalism allows a substantial amount of ambiguity in the grammar.

The `date` rule is also associated with a rewrite statement enclosed in curly brackets. It specifies that a date, e.g. `13/05/2001`, should be expanded into the string *the thirteenth of May two thousand and one*. The symbols `$D`, `$M` and `$Y` are coreferences between constituents of the right-hand side of a rule and the associated rewrite statement. The functions `exp_ordinal()` and `exp_year()` are *rewrite rules* that specify how a constituent should be rewritten in the normalisation process. The grammar may also insert new tokens (as it does `"the"` and `"of"`) as well as reorder or duplicate existing ones.

Parsing rules can optionally be associated with left and right context restrictions. For example, the rule `A → B / C D / E` will expand the sequence `C D` to `A` only if it is preceded by a match of regular expression `B` and followed by a match of regular expression `E`.

The *rewrite rules*, formulated in a two-level regular calculus, do not introduce any novel constructs, so they are not discussed in detail.

The expressive power of the formalism is restricted to a regular language by a) excluding recursion of the form $X \rightarrow \ldots \rightarrow X$[3] and b) an implicit treatment of constituent reordering and duplication, which are non-regular operations. The grammar is converted to an FST translating input strings to possible constituent bracketings including special markers for the non-regular operators. At runtime, the selected bracketing is converted to a tree, on which the extra operations are performed. The constituents of the reordered tree are then sent to the rewrite grammar according to the rewrite rules specified in the rewrite statements of the respective parsing rules.

The grammar FST is typically ambiguous, i.e. it encodes a non-functional rational relation, and may return more than one analysis for an input string. Thus, the actual challenge is to devise a disambiguation strategy with a simple and user-friendly semantics that would make it easier for the grammar developer to maintain control over the behaviour of the grammar.

4 Finite-State Parsing Preferences

As a first step, we shall see how to incorporate parsing preferences into a finite-state grammar. It turns out that preferences can be a) associated with regular operators, and b) translated into an FST in a meaningful way. The result is a prioritisation of alternative paths for a word w in the FST that also corresponds to the order of results in a naïve depth-first search with backtracking. This result will be used in section 5 to establish a deterministic and fail-safe best-first search technique.

4.1 Regular Operators

In a finite-state grammar, complex structures are created from simpler regular expressions using *regular operations*. Their basic inventory comprises *concatenation* (AB, $A \cdot B$), *union* ($A|B$) and *closure* (A^*). Further commonly used operators, such as $?$

[3] This restriction might appear harsh, but text normalisation tasks typically do not require more expressive power (Sproat 1996).

and $+$, can be derived from the primitive ones listed above and need not be considered at first.

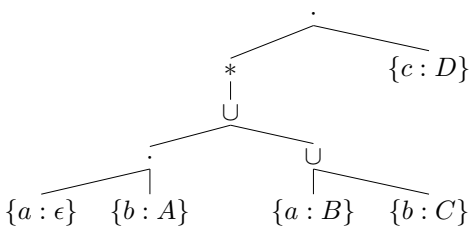

Figure 1: Regular expression $(\{ab : A\}|\{a : B\}|\{b : C\})^*\{c : D\}$ shown as a tree.

The resulting structure forms a tree as shown in figure 1. Such a tree can be compiled into a (non-deterministic) FST. The simplest compilation algorithm (*Thompson's algorithm*) is to create a network of transitions that directly correspond to a traversal of the regexp tree (Hopcroft, Motwani and Ullman 2001). Each node/leaf in the tree translates into two FST states: an *in* state ("we enter the subexpression rooted in the current node") and an *out* state ("we leave the subexpression rooted in the current node"). The *in* state and the *out* state of the root node are, respectively, the initial and the (only) final state of the FST. The states are connected by transitions in accordance with the semantics of the regular operators at the respective tree nodes, as shown in table 1.

Node type	Transitions added
$X = Y\|Z$	$\{X_{in} \overset{\epsilon:\epsilon}{\Rightarrow} Y_{in}, X_{in} \overset{\epsilon:\epsilon}{\Rightarrow} Z_{in}$ $Y_{out} \overset{\epsilon:\epsilon}{\Rightarrow} X_{out}, Z_{out} \overset{\epsilon:\epsilon}{\Rightarrow} X_{out}\}$
$X = Y \cdot Z$	$\{X_{in} \overset{\epsilon:\epsilon}{\Rightarrow} Y_{in}, Y_{out} \overset{\epsilon:\epsilon}{\Rightarrow} Z_{in}$ $Z_{out} \overset{\epsilon:\epsilon}{\Rightarrow} X_{out}\}$
$X = Y^*$	$\{X_{in} \overset{\epsilon:\epsilon}{\Rightarrow} X_{out}, Y_{out} \overset{\epsilon:\epsilon}{\Rightarrow} X_{out},$ $X_{out} \overset{\epsilon:\epsilon}{\Rightarrow} Y_{in}\}$
$X = \{a : o\}$	$\{X_{in} \overset{a:o}{\Rightarrow} X_{out}\}$

Table 1: The FST compilation of union, concatenation, closure and an atomic regexp.

4.2 Local Parsing Preferences

Now suppose we want to find a translation for a string w licensed by a compiled regular expression. A possible, although naïve, strategy is to explore all paths starting from the initial state until we reach the final state. Whenever we find that we have run into a dead end, we backtrack to the previous choice point and explore another

path from there. As long as we omit ϵ-cycles, the algorithm will eventually terminate. Also, if there is more than one translation, the local search decisions taken at the choice points will influence the result.

A simple but powerful disambiguation strategy can be pursued here:

- longest or shortest match (distinguished by notation) at the closure operator;
- exploring disjunction branches in order of their disjuncts.

These two simple rules give the grammar developer full control over all ambiguity in the system. They are also very intuitive: if longest-match is chosen as the default interpretation for the closure operator, the strategy resembles the evaluation of Perl regular expressions, which most users can be assumed to be familiar with.

The reader may object that this disambiguation strategy is *too* simplistic, especially compared to frameworks such as Optimality Theory (OT, (Ellison 1994, Karttunen 1996, Eisner 2000)), which splits linguistic knowledge into a device generating all possible analyses (Gen) and a number of constraints that rank these analyses according to the number and severity of constraint violations. The analysis with the fewest violations wins.

However, this elegant framework is intended as a tool for linguistically adequate theoretical modelling of clear-cut phenomena. It is doubtful that a "dirty" task like text normalisation could be decomposed into a neat hierarchy of constraints filtering the possible analyses. In addition, TTS grammar developers are more likely to be familiar with Perl regular expressions than Optimality Theory. As a matter of fact, the author is not aware of a single large-scale NLP system based on OT.

Nevertheless, we will see that some formal concepts developed in the framework of OT turn out to be very useful for the purpose of preference-based compilation, cf. section 4.4.

4.3 Preference-Driven Search

The simple disambiguation strategy outlined in the previous section can be easily translated to an FST framework by adding some control information to the choice points. Note that only two types of regexp tree nodes introduce non-determinism: union and closure. Thus, if X is a disjunction node and Y, Z are the disjuncts, then the compilation algorithm will create the ϵ-transitions shown in figure 2.

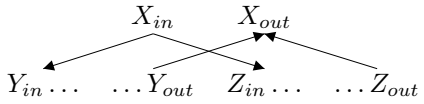

Figure 2: Compilation of the union operator.

The choice point occurs at X_{in}: we can either go to Y_{in} or to Z_{in}. All we need to implement the chosen strategy is to make sure that the move $X_{in} \stackrel{\epsilon}{\Rightarrow} Y_{in}$ will be performed first (i.e. before $X_{in} \stackrel{\epsilon}{\Rightarrow} Z_{in}$).

Figure 3 shows the other case, namely the closure operation. Here, X is the node corresponding to the closure operator, Y the root of the embedded expression and V the parent node of X. The choice point occurs at state X_{out}, from where we can go either to Y_{in} or to V. Note that the former option corresponds to the longest-matchand the latter to the shortest-match strategy.

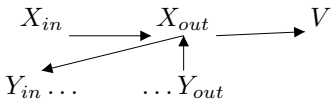

Figure 3: Compilation of the closure operator.

4.4 Encoding

In order to eliminate the non-determinism of the search method sketched above, it is worthwhile to take a closer look at finite-state approaches to Optimality Theory (OT), where a non-deterministic FST (Gen) encodes all possible pairs of inputs (called underlying representations, UR) and outputs (called surface representations, SR). The generated SRs are evaluated by applying a sequence of constraint FSTs $(C_1, \ldots, C_n)$, which either filter out some of the possible SRs or insert *markers* denoting the fulfillment/violation of a constraint. The markers can be used to guide the search for the optimal solution.

Out of all types of OT constraints, the strategy described in the previous section resembles *left-to-right directional constraints* (Eisner 2000). Following Eisner, whenever two possible transitions labelled with the same input symbol $a \in \Sigma \cup \{\epsilon\}$ leave a state q, the preferred one is assigned the mark m_0 and the less preferred one the mark m_1. In the case of the compilation method specified in section 4.1, such choice points are always binary and are possible only for $a = \epsilon$. Thus, the simplest encoding is to set $e.output = m_0$ or respectively $e.output = m_1$ for all such ϵ-transitions leaving choice point states.

In this way, each path $\pi = e_1, \ldots, e_n$ in a transducer T can be associated with a sequence of preference marks $\omega = \omega_1 \ldots \omega_k$, which can be extracted from $\pi.output$ simply by ignoring all output symbols other than m_0 and m_1. Let such a sequence be denoted $\pi.score$.

Note that the relative preference order of two paths $\pi^{(1)}$ and $\pi^{(2)}$ is expressed by the lexicographic order of the mark sequences:

$$\pi^{(1)} \prec \pi^{(2)} \iff \pi^{(1)}.score <_{lex} \pi^{(2)}.score$$

Generalised to sets Π of paths accepting a string w, this criterion states that the preferred path is the first one in lexicographic order:

$$\pi_{best} = min_{\prec}(\Pi)$$

In particular, if $\Pi_T(w)$ denotes the set of all paths in T that consume $w \in \Sigma^*$, then the preferred translation for w is the output of the path $\pi_{best}(w) \in \Pi_T(w)$ defined as:

$$\begin{aligned}\pi_{best}(w) \quad = \quad & min_{\prec}(\{\pi \in \Pi_T(w) : \\ & \pi.source = q_0 \wedge \pi.target \in F\})\end{aligned}$$

Note that we may want to make transducer T ϵ-free for further processing stages. This is not a problem because the preference order of paths is preserved in ϵ-elimination.

5 Search Strategy

Let $\hat{T} = (\Sigma, \Delta, \hat{Q}, q_0, \hat{E}, \hat{F})$ be an ϵ-free FST constructed from a regular expression using the compilation method described in section 4.1 and some ϵ-elimination algorithm. Given an input string $w = a_1 \dots a_t$, we want to find the best-scoring successful path for w in $\hat{T}$.

Note that if the unsuccessful paths are pruned away in advance, the search boils down to a chain of purely local decisions. We start by choosing the lowest-score arc (q_0, a_1, q_1, o_1) starting in q_0, and then choose the lowest-score transition starting in q_1 and accepting a_2, etc.

The set of all successful paths for w can be determined in time $O(t)$ using the following factorisation method. Let $\hat{E}_{DFSA} = \{\langle q, a, q' \rangle : \exists o \in \Delta^* : \langle q, a, q', o \rangle \in \hat{E}\}$ be the projection of the transitions of $\hat{T}$ onto the first three components (i.e. source state, input symbol and output symbol). We determinize the NFSA $A = (\Sigma, \hat{Q}, q_0, \hat{E}_{DFSA}, \hat{F})$ for the input language of $\hat{T}$ using a variant of the subset construction algorithm (Hopcroft et al. 2001). This operation yields a) a DFSA $\overrightarrow{A} = (\Sigma, \overrightarrow{Q}, \overrightarrow{q}_0, \overrightarrow{\delta}, \overrightarrow{F})$ accepting $\mathcal{L}(\hat{T})$ and b) a function $\overrightarrow{h} : \overrightarrow{Q} \rightarrow 2^Q$ mapping each state of the DFSA to the corresponding set of states of $\hat{T}$. More precisely, $\overrightarrow{h}(\overrightarrow{\delta}(\overrightarrow{q}_0, u))$ is the set of states reachable in $\hat{T}$ from q_0 by consuming the string u.

In a similar way, we can construct an acceptor $\overleftarrow{A} = (\Sigma, \overleftarrow{Q}, \overleftarrow{q}_0, \overleftarrow{\delta}, \overleftarrow{F})$ and a function $\overleftarrow{h}$ by reversing and determinising A. Accordingly, $\overleftarrow{h}(\overleftarrow{\delta}(\overleftarrow{q}_0, v^{-1}))$ is the set of all states $q \in \hat{Q}$ such that $q \stackrel{v}{\Rightarrow} F$.

If now $w = uv \in \Sigma^*$, then the intersection $\overrightarrow{h}(\overrightarrow{\delta}(\overrightarrow{q}_0, u)) \cap \overleftarrow{h}(\overleftarrow{\delta}(\overleftarrow{q}_0, v^{-1}))$ is the set of all states on a successful path in $\hat{T}$ accepting v after consuming the prefix u. Thus, given a string $w = a_1 \dots a_t$, we can determine the successful paths in $\hat{T}$ in time $O(t)$ by:

- running $\overrightarrow{A}$ on string w and keeping the path $\overrightarrow{q}_0 \overrightarrow{q}_1, \dots, \overrightarrow{q}_t$;
- running $\overleftarrow{A}$ on w^{-1} and keeping the path $\overleftarrow{q}_0 \overleftarrow{q}_1, \dots, \overleftarrow{q}_t$;
- forming a sequence of *reachability sets* $R_0, \dots, R_t$ constructed as follows: $R_j = \overrightarrow{h}(\overrightarrow{q}_j) \cap \overleftarrow{h}(\overleftarrow{q}_{t-j})$, $j = 0, \dots, t$.

Since T is non-deterministic, there are several alternative paths $\pi^{(i)} = e_1^{(i)}, \dots, e_t^{(i)}$ for w, each associated with a unique score $\pi^{(i)}.score$. The preferred

translation is the output of the first path $\pi^{(i)}$ according to the lexicographic order of the path scores. Thus, the preferred path can be found deterministically in time $O(t)$ according to the following formula:[4]

$$e_j = \begin{cases} min_{\prec}(Out(q_0, a_1)) & j = 1 \\ min_{\prec}(Out(e_{j-1}.source, a_j)) & j > 1 \end{cases} \quad (1)$$

In this way, we arrive at a deterministic parsing strategy that guarantees finding the best parse (according to the locally expressed disambiguation preferences) in time $O(t)$.

The construction of $\overleftarrow{A}$ and $\overrightarrow{A}$ together with a recursive formula for the best path is closely related to *bimachines* (Berstel 1979, Roche and Schabes 1996). A *bimachine* is a triple $B = (\overrightarrow{A}, \overleftarrow{A}, \gamma)$, where $\overrightarrow{A} = (\Sigma, \overrightarrow{Q}, \overrightarrow{q}_0, \overrightarrow{\delta}, \overrightarrow{Q})$ is a left-to-right DFSA, $\overleftarrow{A} = (\Sigma, \overleftarrow{Q}, \overleftarrow{q}_0, \overleftarrow{\delta}, \overleftarrow{Q})$ is a right-to-left DFSA, and $\gamma : \overrightarrow{Q} \times \overleftarrow{Q} \to \Delta^*$ the output function of B. Applied to a string $w = a_1 \ldots w_t$, B outputs the string $b_1 \ldots b_t$, where $b_i = \gamma(\overrightarrow{\delta}^*(\overrightarrow{q}_0, b_1 \ldots b_{i-1}), \overleftarrow{\delta}^*(\overleftarrow{q}_0, b_t \ldots b_i))$.

The importance of bimachines lies in the fact that a) they are deterministic and b) every unambiguous FST — even a non-determinisable one — can be converted to a bimachine. Since most interesting cases of FSTs actually *are* ambiguous, the construction presented in this section extends the notion of bimachines to FSTs which are disambiguated at runtime via preference marks.[5] Accordingly, the output function is not specified explicitly; instead, it follows from the recursive formula (1).

6 Optimisation and Evaluation

6.1 Runtime Optimisation

The algorithm can be made more efficient by means of precompilation. Instead of running $\overrightarrow{A}$ and $\overleftarrow{A}$ separately, and then performing the disambiguation step described by (1), the task can be performed in only two stages.

In the first pass, the acceptor $\overleftarrow{A}$ is run on the reversed input string, producing a path $\overleftarrow{q}_0, \ldots, \overleftarrow{q}_t$. This path is then combined with the original input w in order to form a sequence of state-input pairs of the following form: $\langle a_1, \overleftarrow{q}_{t-1}\rangle, \ldots, \langle a_j, \overleftarrow{q}_{t-j}\rangle, \ldots, \langle a_t, \overleftarrow{q}_0\rangle$. This sequence serves as input to the second component of the system, which is a sequential transducer $\tilde{T} = (\Sigma \times \overleftarrow{Q}, \Delta, \hat{Q}, q_0, \delta, \sigma, \hat{F})$ over the complex input alphabet $\Sigma \times \overleftarrow{Q}$. If w is accepted, $\tilde{T}$ outputs the preferred translation of w. The functions δ and σ are defined as follows.

$$\begin{aligned} \delta(q, \langle a, q'\rangle) &= min_{\prec_{(q,a)}}(\{r \in \overleftarrow{h}(q') : q \stackrel{a}{\Rightarrow}_{\hat{T}} r\}) \\ \sigma(q, \langle a, q'\rangle) &= o(q, a, \delta(q, (a, q'))). \end{aligned}$$

The application of the above device to a string w is deterministic and very efficient: it requires $2|w|$ transition lookup steps; the execution time of each of these steps can be made constant by employing appropriate data structures.

[4] $Out(q, a)$ denotes the set of all transitions leaving q via symbol a. $min_{\prec}$ is well-defined because the transitions $e \in Out(e_{j-1}.source, a_t)$ have distinct scores, i.e., they are totally ordered by $\prec$.

[5] For another application of the concept of bimachine factorisation to ambiguous FSTs, see Kempe (2001).

6.2 Compile-Time Optimisation

There are several possible optimisations aiming at speeding up the compilation process. First of all, the structure $(\overleftarrow{A}, \tilde{T})$ introduced above for runtime optimisation is also faster to compile since its compilation involves only one potentially expensive determinisation step (for $\overleftarrow{A}$) instead of two (for $\overleftarrow{A}$ and $\overrightarrow{A}$).

Another improvement is related to the representation of the markers, which need not be present at runtime. Instead, alternative transitions can be stored in a list in an order corresponding to the relation $\prec$.

6.3 Compilation Speed

As mentioned above, the determinisation of $\overleftarrow{A}$ is the only expensive step in the construction. It took 28 minutes for a grammar fragment comprising 78 rules, while the running time of the remaining compilation steps was under 2 seconds. For larger grammar fragments, the discrepancy was even bigger, indicating scalability problems. Therefore, the decision was taken to introduce two compilation modes: *development* and *release*.

In the development mode, $\overleftarrow{A}$ is not constructed, and the search for the optimal parse involves backtracking. On average, this is around 11 times slower than deterministic search, but the difference is not noticeable to the grammar developer. The benefit is that a grammar can be developed, compiled and tested interactively.

In the release mode, the construction of $\overleftarrow{A}$ is sped up by removing some irrelevant bracketing information from the FST. For the above test fragment, the size of the grammar transducer was thus reduced from 110,056 to 29,231 transitions, and $\overleftarrow{A}$ took only 8 minutes to construct. For the largest grammar available, comprising 214 rules, compilation took 34 minutes, resulting in an FST containing 104,551 transitions (after reduction). $\overleftarrow{A}$ had 671,331 transitions.

The above behaviour of the compiler means it is scalable to medium-sized systems comprising hundreds of rules. For even larger grammars, we contemplate the construction of several preference-based bimachines: one bimachine recognising the matches of the top-level grammar rules (e.g. `date`, `time` or `phone_number`), and one for each of the respective subgrammars.

6.4 Expressive Power

The formalism presented in this paper is a compromise between expressive power and processing efficiency: linear-time parsing is achieved at the expense of keeping the formalism relatively simple. The question is how much the restrictions imposed on the grammar affect the convenience of grammar development.

As far as text normalisation in TTS systems is concerned, the compromise pays off to a large extent. The predecessor of our system, which employed exactly the same disambiguation strategy (longest match plus a left-to-right preference order on disjunctive rules), proved to be expressive enough as a formalism for industrial-scale

multilingual development.[6] The grammar developers have largely found the parsing strategy intuitive and easy to follow — despite some criticism expressed with regard to the global ordering of rules, which was deemed too rigid in certain cases. Hence, the development of larger grammar fragments (partly using automatic conversion from the old grammar formalism), can be expected to be straightforward.

7 Extensions

In section 4.1, the inventory of regular operators was restricted to the primitive ones: union, concatenation and closure. Obviously, the disambiguation semantics can be extended to other, derived operators such as `R?` or `R+` by setting the transition scores accordingly on the choice points introduced by these operators.

The user may also wish to retain some ambiguity, using the parser for filtering rather than finding one optimal solution. In such a case, preference marks are not inserted at the affected nodes of the regular expression. The search algorithm requires a straightforward adaptation to the case of multiple optimal paths.

8 Related Work

Discussion of related work is split into three areas: the grammar formalism, expressing preferences in hand-writable rules and the run-time evaluation of such preference-based grammars.

8.1 Formalism

In the area of text-to-speech synthesis, pioneering finite-state work has been done in the framework of weighted FSTs at Bell Labs (Sproat 1996, Mohri and Sproat 1996). There, as in most finite-state approaches to NLP, text processing tasks are viewed as successive stages of string rewriting, each implemented by an FST; the FSTs for different stages may be combined via FST composition.

In the formalism presented here, a similar idea underlies the *rewrite rules*, formulated in a regular calculus. However, there are also the more structure-oriented *parsing rules*, which establish a tree-shaped derivation for the constructs being normalised. The possibility of producing such derivations has proved to be of great help to the grammar developers.

8.2 Parsing Preferences

There are two basic ways of expressing parsing preferences in a finite-state framework. One is to encode them as real-valued *weights* associated with the transitions of a weighted FST (Mohri 1997). The transducer is ambiguous, but adding weights along alternative paths yields a preference order over the possible parsing results. Thus, the weights may be used to guide the search for the optimal solution. This framework is

[6] The languages covered comprised English, Greek, German, Spanish and French; each of the language-specific grammars consisted of up to one thousand rules.

particularly well-suited for probabilistic NLP approaches where the weights express probabilities (typically as logarithms).

The other way of handling preferences is to apply them at compile time, when particular rules and constraints are combined to form a grammar system. The resulting FST contains only the optimal paths; hence it is unambiguous and can be either determinised or converted to a bimachine.

A possible method of combining prioritised rules is to express each of them as an FST and then join these FSTs via an operation called *priority union* (Karttunen 1998). The priority union of of two FSTs T_1 and T_2 is defined as $T_1 \cup (T_2 \circ (\overline{Upper(T_1)}))$, where $\overline{L}$ is the complement of a regular language L and $Upper(T)$ is the language accepted by transducer T. Priority union restricts transducer T_2 to the complement of T_1. As a result, all conflicts between translations in T_1 and T_2 are resolved in favour of T_1 while strings not accepted by T_1 can still be rewritten by T_2. Similar formalisations have also been given for the longest-match and shortest-match semantics of regular expressions (Karttunen 1996, Gerdemann and van Noord 1999).

The main difficulty related to this "algebraic" approach is that it involves repeated application of costly operations such as regular complement (exponential in $|Q|$) and composition (quadratic in $|Q|$). The resulting FST is non-deterministic, hence another worst-case exponential determinisation step (subset construction or bimachine creation) is required. All this may lead to very slow compilation for realistic text normalisation grammars. In order to estimate the processing overhead caused by these operations, we replaced all instances of regular union in the grammar by priority union. As a result, the compilation time (before determinisation) for the 78-rule grammar evaluated in section 6.3 increased from below 2 seconds to 94 seconds.

One might argue that the running time of these operations is of secondary importance as compilation can be done off-line. However — as already mentioned in section 1 — compilation times exceeding one minute may seriously hamper the productivity of grammar development.

This is the reason why we decided to employ a compilation method that does not eliminate non-optimal paths from the transducer at compile time, but associates its transitions with additional control information (preference markers).[7] From this perspective, our compilation method exhibits a strong resemblance to to the weighted FST paradigm: ambiguity is left in the FST and resolved at runtime using dynamic programming. However, the markers embedded into the output strings of the FST are evaluated in a completely different way than real-valued weights are.

The construction of the FST encoding all the rules is cheap. If N is the number of regular operators and occurrences of symbols in the grammar, it involves a) Thompson construction leading to the creation of an FST with $|Q| = O(N)$ states and $|E| = O(N)$ transitions in time $O(N)$ and b) ϵ-elimination, which can be done in time $O(|Q| \cdot |E|)$. Hence, the construction of the grammar FST is bounded by $O(N^2)$. The observed dependency between N and the actual compilation time is linear. The only costly operation is the creation of the reverse acceptor $\overleftarrow{A}$, which is omitted in the

[7] Still, an unambiguous FST can always be recovered from an FST with preference marks, e.g. using the *directional best paths* algorithm (Eisner 2000, Eisner 2002) designed to implement directional constraint evaluation in the OT framework.

development mode.

8.3 Search for the Optimal Solution

The standard solution to the problem of finding the best path for a string w in an ambiguous FST is to compose the FST with an FSA encoding the input string. If the scores associated with the transitions are viewed as edge weights of a directed graph, finding the best path in the resulting FST is then an instance of the single-source shortest path problem. Viterbi search performs both the composition (creating a structure called a *trellis*) and search for the optimal solution within this structure in time $O(|w|)$.

This method can also be used to find the best-scoring path in the case of our construction. However, parsing would most certainly be less efficient than in the case of a bimachine due to the overhead caused by the construction of the trellis and keeping up to $|Q|$ best path candidates for each string position, which is obviously more expensive than $2 \cdot |w|$ lookups in the transition table (i.e. the total parsing cost in the case of a bimachine).

9 Conclusion

The rule compiler described in this paper presents an alternative to both traditional weighted FST compilation (Mohri and Sproat 1996) and the "algebraic" approach in the vein of Kaplan and Kay (1994). Ambiguity is dealt with using a simple but powerful disambiguation strategy that may be viewed as a mixture of priority union (Karttunen 1998) and longest/shortest match (Gerdemann and van Noord 1999). The use of preference markers makes it possible to avoid the relatively costly operations typically associated with the compilation of the above constructs. On the other hand, the resulting structure (an ambiguous FST containing preference markers and a right-to-left deterministic acceptor) still makes it possible to conduct deterministic search for the optimal result. In this way, the compiler combines fast compilation and efficient processing at runtime.

References

Berstel, J.(1979), *Transductions and Context-Free Languages*, Teubner Verlag.

Eisner, J.(2000), Directional constraint evaluation in Optimality Theory, *Proceedings of COLING 2000*, Saarbrücken, Germany, pp. 257–263.

Eisner, J.(2002), Comprehension and compilation in Optimality Theory, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia.

Ellison, M.(1994), Phonological derivation in optimality theory, *COLING 1994*.

Gerdemann, D. and van Noord, G.(1999), Transducers from rewrite rules with back-references, *Proceedings of EACL 99*.

Hopcroft, J. E., Motwani, R. and Ullman, J. D.(2001), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley.

Hunt, A. and Black, A.(1996), Unit selection in a concatenative speech synthesis system using a large speech database, *Proceedings of ICASSP 96*, Vol. 1, Atlanta, Georgia, pp. 373–376.

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

Karttunen, L.(1996), Directed replacement, *in* A. Joshi and M. Palmer (eds), *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics*, Morgan Kaufmann Publishers, San Francisco, pp. 108–115.

Karttunen, L.(1998), The proper treatment of optimality in computational phonology, *in* L. Karttunen (ed.), *International Workshop on Finite State Methods in Natural Language Processing*, Association for Computational Linguistics, Somerset, New Jersey, pp. 1–12.

Kempe, A.(2001), Factorization of ambiguous finite-state transducers, *Lecture Notes in Computer Science*.

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

Mohri, M. and Sproat, R.(1996), An efficient compiler for weighted rewrite rules, *Meeting of the Association for Computational Linguistics*, pp. 231–238.

Roche, E. and Schabes, Y.(1996), Introduction to finite-state devices in natural language processing, *Technical report*, Mitsubishi Electric Research Laboratories, TR-96-13.

Sproat, R.(1996), Multilingual text analysis for text-to-speech synthesis, *Workshop on Extended Finite State Models of Language (ECAI '96), August 12-16*, von Neumann Society of Computer Science, Budapest, Hungary.