Finite State Transducers with Predicates and Identities

Gertjan van Noord and Dale Gerdemann

Abstract. An extension to finite state transducers is presented, in which atomic symbols are replaced by arbitrary predicates over symbols. The extension is motivated by applications in natural language processing (but may be more widely applicable) as well as by the observation that transducers with predicates generally have fewer states and fewer transitions.

Although the extension is fairly trivial for finite state acceptors, the introduction of predicates is more interesting for transducers. It is shown how various operations on transducers (e.g. composition) can be implemented, as well as how the transducer determinization algorithm can be generalized for predicate-augmented finite state transducers.

1. Introduction

Finite automata are widely used in natural language processing. We present an extension to finite automata, in which atomic symbols are replaced by arbitrary predicates over symbols. Although the extension is fairly trivial for finite state acceptors, the introduction of predicates is more interesting for transducers. Below, we show how various operations on such extended acceptors and transducers can be defined and implemented. But first the extension is motivated as follows.

1.1. Predicates

In natural language processing, it is often more natural to think of symbols in terms of predicates or classes. The linguistic principle of *Community* dictates that similar segments behave similarly. Predicates are a means to express this similarity. In computational phonology it is thus more natural to talk about *vowels* and *consonants* rather than enumerate each of the phonemes in these classes. Phonological generalizations typically refer to predicates such as *fricative*, *nasal*, *voiced* and very seldomly to individual phonemes directly. Therefore, in finite state computational phonology, some have proposed finite state automata in which transitions are associated with sets of symbols (Walther, 1999; Bird and Ellison, 1994; Eisner, 1997; Walther, 2000).

As a further piece of motivation for the introduction of predicates, consider the *unknown symbol* regular expression operator, typically written `?`, as it is available in some regular expression compilers (Karttunen et al., 1996; van Noord and Gerdemann, 1999). An obvious implementation will expand the `?` operator into a set of transitions for each of the symbols in the alphabet Σ. In our proposal, the `?` operator will be expanded into a single transition with an associated predicate which is true for all symbols; this has the advantage that Σ need not be explicitly defined. As a consequence, there is no need to assume that the alphabet is finite. Such considerations become important for applications with large alphabets, such as the Unicode alphabet. Even larger alphabets may surface in natural language processing applications in which the symbols are words. Typical electronic dictionaries have at least 200K words and even this large size alphabet is not enough to handle unrestricted texts. Realistically, robust syntactic parsing requires an infinite alphabet.[1]

Below, we define predicate augmented finite state automata more precisely; for now it suffices to assume that such automata are similar to classical finite state automata, except that we have predicates instead of symbols.

1.2. Notation

The predicates used in this paper are predicates on Σ. So, each predicate π is a total function such that for each $\sigma \in \Sigma$, $\pi(\Sigma)$ is either *true* or *false*. If π is the *characteristic function* of the set $S \subseteq \Sigma$, i.e., $S = \{\sigma \in \Sigma | \pi(\sigma)\}$, then in transition diagrams we often write S instead of π. As usual, if S is a set, then the complement of S is written $\overline{S}$. Moreover, if S is of the form $\{\texttt{c}\}$, i.e., a singleton set, then we abbreviate this predicate simply as `c`. As a special case, Σ is written as `?`. In transducers, a transition is associated both with an input predicate π_d as well as with an output predicate π_r; such a pair of predicates is written as $\pi_d : \pi_r$.

Below, we will often refer to states in automata using p, q, and r. For examples of symbols we use characters from the beginning of the alphabet in typewriter font such as `a, b, c`; for sequences of symbols we use characters w, x, y, z. Typically, we use σ as a variable that takes a symbol as its value. Examples of predicates are written in small caps, using characters from the beginning of the alphabet, like A, B, C. A variable that takes a predicate as its value is written π. A sequence of predicates is often written using Greek symbols ϕ, ψ. Finally, note that the empty sequence is written ϵ, for either the empty sequence of symbols or the empty sequence of predicates.

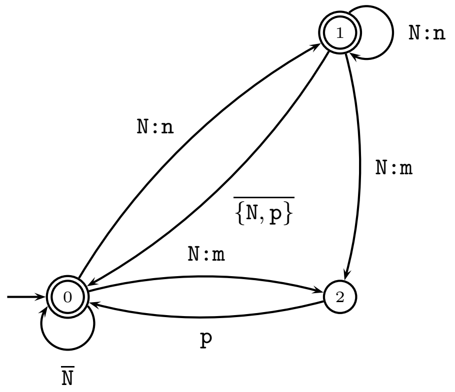
[1] If infinite alphabets are allowed, then certain non-regular languages such as $\{0, 1, \ldots\}^*$ can be recognized. A similar generalization of regular languages is used by (Perrin, 1990).

1.3. Identities

Consider the following phonological rule (from (Karttunen, 1991)) in which an underlying nasal segment N is mapped either to an m (if followed by a p) or an n:

$$\mathtt{N} \rightarrow \mathtt{m}/\_\ \mathtt{p}; \quad \text{elsewhere } \mathtt{n}$$

A transducer implementing this phonological rule can be illustrated as follows:



This transducer contains a single start state 0, and two final states, 0 and 1. First consider the cyclic transition on state 0, labeled by the predicate $\overline{\mathtt{N}}$, i.e., the predicate which is true of all symbols except the symbol N. As long as the transducer does not read this special nasal segment N, it remains in state 0 and simply copies its input. Upon reading an N, the transducer non-deterministically moves to state 1 or state 2, writing out an n or m respectively. In the first case, the next input symbol cannot be a p; in the second case the next input symbol must be a p.

Note that the transition from state 2 to state 0 simply contains a p. The idea here is that if the input and output symbol must be identical, only a single predicate is written for that transition. The same abbreviation is used for the transition from 1 to 0, as well as over the looping transition from 0 to 0 with predicate $\overline{\mathtt{N}}$. The intention here of course is that every incoming segment which is not equal to N should be mapped to itself in the output. However, note that this is quite different from the pair of predicates $\overline{\mathtt{N}} : \overline{\mathtt{N}}$. The latter would map an incoming symbol to an arbitrary output symbol, as long as both symbols are unequal to N.

The example illustrates an important point: if predicates are introduced in transducers, then for typical examples we must also be able to express the identity of input and output of a transition. In this example, if there were no way to express the identity between input and output, then we would be forced to have multiple transitions such as `a:a, b:b, c:c, d:d` for all of the relevant symbols; the introduction of predicates can be exploited in transducers only if identity between input and output can be expressed as well.

Expressing identity between input and output is crucial. This notion of identity can be seen as a consequence of the linguistic principle of *Faithfulness*: corresponding input and output segments tend to be identical. A similar argument is expressed in (Gildea and Jurafsky, 1996). Indeed, many interesting transducers are of the type 'change all occurrences of α in some specific context into β, and pass on the rest of the input unaltered'. The various replacement and 'local extension' operators all produce transducers of this kind (Karttunen, 1995; Roche and Schabes, 1995; Karttunen, 1996; Kempe and Karttunen, 1996; Gerdemann and van Noord, 1999a). Identities can be seen as a limited case of *backreferencing*. Backreferencing is an extension of regular expressions widely used in editors, scripting languages and other tools. A limited version of finite-state calculus backreferences is discussed in (Gerdemann and van Noord, 1999b).

1.4. Smaller Automata

Another motivation for the introduction of predicates is the observation that the resulting automata are smaller. The size of automata is an important problem in practice (Daciuk, 1998; Kiraz, 1999). With predicates, potentially large sets of transitions are replaced by a single transition. For example, if an automaton has transitions from state p to state q over all ASCII symbols except for a symbol `a`, for which there is a transition from p to r, then there are 128 transitions leaving p. Using predicates, there are only two transitions leaving p (one labeled by a predicate $\{\mathtt{a}\}$, and one labeled by $\overline{\{\mathtt{a}\}}$). But note that similar space reductions can be achieved using *failure transitions* and related techniques (Kowaltowski et al., 1993; Kiraz, 1999; Daciuk, 2000; Klarlund, 1998).

More interesting space reductions can be achieved in the case of transducers. The introduction of predicates with identity not only leads to transducers with fewer transitions, but also to transducers that have fewer states. This observation will be discussed in section 3.7. In section 4.2 we show that this space reduction is achieved for linguistically relevant examples too.

The implementation of various operations is faster for smaller automata. Although the implementation of some of the relevant operations becomes somewhat more complex, it is our experience that in almost all cases overall performance improves considerably.

1.5. Determinization of non-functional transducers

We show below that the introduction of predicates has the interesting effect that certain non-functional transducers can be treated by the transducer determinization algorithm (Oncina et al., 1993; Reutenauer, 1993; Mohri, 1996; Roche and Schabes, 1995; Roche and Schabes, 1997). Therefore a larger class of transductions can be implemented efficiently.

1.6. Previous Work

A possible implementation of the question mark operator is the introduction of a special symbol **?** in finite state automata.[2] This special symbol is understood as 'any alphabet symbol not mentioned in the automaton', in order to translate examples such as `?-a`. This technique requires that each question mark operator is expanded into the set of symbols occurring in the regular expression as a whole. This solution (implemented in a previous version of the FSA Utilities (van Noord and Gerdemann, 1999) and in `xfst`, the Xerox regular expression compiler (Karttunen et al., 1996)) therefore leads to a proliferation of transitions. For example, the expression $(a..z{\cdot}? - d)$ would result in an automaton with 52 transitions: 26 transitions from the initial state to an intermediate state for each of the letters of the alphabet and 26 transitions from this intermediate state to a final state for each of the letters except 'd', as well as for **?**.[3]

The idea to allow predicates on transitions instead of symbols is also mentioned in (Watson, 1999a) and (Watson, 1999b). The details of this proposal, however, are not given. Apparently, in Watson's proposal predicates potentially inspect arbitrary parts of the input, and consume arbitrary prefixes of the input; the resulting formalism is therefore much more powerful, and hence various closure and efficiency properties are not applicable. In contrast, for the type of predicates proposed here, these attractive properties in fact are applicable, as is shown in the remainder of the article.

[2] Note that in such an implementation, the regular expression operator ? (any symbol) is not to be confused with the special symbol in automata **?** (any symbol not occurring in the automaton).

[3] Here we assume that we are not explicitly representing states which are not co-accessible, i.e. for which there is no path to a final state.

1.7. Overview

In the next section, predicate-augmented finite state recognizers are introduced, and it is shown how various operators and algorithms can be generalized. In section 3 predicate-augmented finite state transducers are introduced. We show that operations such as composition can be implemented straightforwardly; in addition we show how the transducer determinization algorithm can be generalized. The generalization leads to the definition of predicate-augmented finite state transducers with a bounded queue; the queue is required to be able to treat identities correctly. It is shown that this device allows a more compact representation of some finite-state transductions than the classical model. In section 5 we discuss some open problems and directions for future research.

2. Finite State Recognizers with Predicates

2.1. Definition

A predicate-augmented finite state recognizer (pfsr) M is specified by $(Q, \Sigma, \Pi, E, S, F)$ where Q is a finite set of states, Σ a set of symbols, Π a set of predicates over Σ, E a finite set of transitions $Q \times (\Pi \cup \{\epsilon\}) \times Q$. Furthermore, $S \subseteq Q$ is a set of start states and $F \subseteq Q$ is a set of final states.

The relation $\widehat{E} \subseteq Q \times \Sigma^* \times Q$ is defined inductively:

1. for all $q \in Q$, $(q, \epsilon, q) \in \widehat{E}$,
2. for all $(p, \epsilon, q) \in E$, $(p, \epsilon, q) \in \widehat{E}$,
3. for all $(q_0, \pi, q) \in E$ and for all $\sigma \in \Sigma$, if $\pi(\sigma)$ then $(q_0, \sigma, q) \in \widehat{E}$
4. if (q_0, x_1, q_1) and (q_1, x_2, q) are both in $\widehat{E}$ then $(q_0, x_1 x_2, q) \in \widehat{E}$

The language $L(M)$ accepted by M is defined to be $\{w \in \Sigma^* | q_s \in S, q_f \in F, (q_s, w, q_f) \in \widehat{E}\}$.

A pfsr is called ϵ-free if there are no $(p, \epsilon, q) \in E$. For any given pfsr there is an equivalent ϵ-free pfsr. It is straightforward to extend the corresponding algorithm for classical automata. Without loss of generality we assume below that pfsr are ϵ-free.

2.2. Properties

It is clear that in the case of recognizers, the addition of predicates is of limited theoretical interest. Let M^c be a classical finite automaton (Q, Σ, E, S, F) with Q a finite set of states, Σ a set of symbols, $S \subseteq Q$ the set of start states, $F \subseteq Q$ the set of final states and E a finite set of transitions $Q \times \Sigma \times Q$. Furthermore, let $s(p, q)$ be the set of symbols on transitions from p to q, i.e., $s(p, q) = \{\sigma | (p, \sigma, q) \in E\}$. If M^c is such a (minimal) finite automaton then clearly the equivalent (minimal) pfsr is given by $(Q, \Sigma, 2^{\Sigma}, E', S, F)$ where $E' = \{(p, s(p, q), q) | (p, s, q) \in E\}$. The construction in the other direction is similar.

The pfsr device typically is more compact in the number of transitions than an equivalent finite automaton. In the worst case, however, the number of transitions is the same (if it is the case for all states that its outgoing transitions have different target states for each symbol). In the best case, the number of transitions is reduced by a factor of $|\Sigma|$.

2.3. Operations on recognizers

Since predicate-augmented finite state recognizers are equivalent to ordinary finite-state automata, the class of languages defined by psfr is closed under the the usual regular operations such as *union*, *concatenation*, *Kleene-closure* and *reversal*. From a practical point of view, however, it is interesting to note that it is trivial to generalize the corresponding constructions for classical finite state automata (cf. for instance (Hopcroft and Ullman, 1979)). This means that the various constructions can be implemented directly, without the need to expand into ordinary finite automata first, which is impractical for large alphabets.

2.3.1. *Intersection*

An important and powerful operation is intersection. In the classical case, an automaton for the intersection of the languages defined by two given automata M_1 and M_2 is constructed by considering the cross product of states of M_1 and M_2. A transition $((p_1, p_2), \sigma, (q_1, q_2))$ exists iff the corresponding transition (p_1, σ, q_1) exists in M_1 and (p_2, σ, q_2) exists in M_2. In the case of pfsr a similar construction can be used, but instead of requiring that the symbol σ occurs in the corresponding transitions of M_1 and M_2, we require that the resulting predicate is the conjunction of the corresponding predicates in M_1 and M_2. The same technique is described in (Walther, 1999).

Given ϵ-free pfsr $M_1 = (Q_1, \Sigma, \Pi, E_1, S_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Pi, E_2, S_2, F_2)$, the intersection $L(M_1) \cap L(M_2)$ is the language accepted by $M = (Q_1 \times Q_2, \Sigma, \Pi, E, S_1 \times S_2, F_1 \times F_2)$ and $E = \{((p_1, q_1), \pi_1 \wedge \pi_2, (p, q)) | (p_1, \pi_1, p) \in E_1, (q_1, \pi_2, q) \in E_2\}$.

2.3.2. *Determinization*

An ϵ-free pfsr is deterministic if there is a single start state, and if for all states $q \in Q$ and symbols $\sigma \in \Sigma$ there is at most one transition (q, π, q') such that $\pi(\sigma)$. If a pfsr M is deterministic then checking whether a given string w is accepted by M can be implemented efficiently: linear in w, and independent on the size of M.

A determinization algorithm (Aho et al., 1986; Hopcroft and Ullman, 1979; Johnson and Wood, 1997) maintains subsets of states. Each subset is a state in the deterministic machine. To compute the transitions leaving a given subset D, a determinization algorithm computes for each symbol $\sigma \in \Sigma$ the set of states Q such that $p \in D, q \in Q$ and $(p, \sigma, q) \in E$.

In the case of predicates, however, transitions might overlap. For example, one transition may be applicable for high vowels, whereas another transition may be applicable for round vowels. In the determinized pfsr, such overlaps are not allowed. Therefore, we create a separate transition for high and round vowels, another transition for vowels which are high but not round, and a third transition for vowels which are round but not high.

In general, in order to compute the transitions leaving a given subset D we do as follows. Firstly we compute the function $\mathit{Trans}^D \colon \Pi \rightarrow 2^Q$, defined as: $\mathit{Trans}^D(\pi) = \{q \in Q | p \in D, (p, \pi, q) \in E\}$. For example, suppose $D = \{p\}$, and suppose we have transitions

$$\begin{aligned} E = \{ \ & (p, \pi_1, q_1), (p, \pi_1, q_2), (p, \pi_2, q_2), (p, \pi_2, q_3), \\ & (p, \pi_2, q_4), (p, \pi_3, q_3), (p, \pi_3, q_5)\} \end{aligned}$$

In that case:

$$\mathit{Trans}^D(\pi_1) = \{q_1, q_2\}, \mathit{Trans}^D(\pi_2) = \{q_2, q_3, q_4\}, \mathit{Trans}^D(\pi_3) = \{q_3, q_5\}$$

Let Π' be the predicates in the domain of Trans^D. For each split of Π' into two subsets $\pi_1 \dots \pi_i$ and $\pi_{i+1} \dots \pi_n$ we have a transition:

$$(D, \pi_1 \wedge \dots \wedge \pi_i \wedge \neg\pi_{i+1} \wedge \dots \wedge \neg\pi_n, \mathit{Trans}^D(\pi_1) \cup \dots \cup \mathit{Trans}^D(\pi_i))$$

for the example we obtain the transitions:[4]

$$\begin{array}{llll}
(& D, & \pi_1 \wedge \pi_2 \wedge \pi_3, & \{q_1, q_2, q_3, q_4, q_5\} &) \\
(& D, & \pi_1 \wedge \pi_2 \wedge \neg\pi_3, & \{q_1, q_2, q_3, q_4\} &) \\
(& D, & \pi_1 \wedge \neg\pi_2 \wedge \pi_3, & \{q_1, q_2, q_3, q_5\} &) \\
(& D, & \pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3, & \{q_1, q_2\} &) \\
(& D, & \neg\pi_1 \wedge \pi_2 \wedge \pi_3, & \{q_2, q_3, q_4, q_5\} &) \\
(& D, & \neg\pi_1 \wedge \pi_2 \wedge \neg\pi_3, & \{q_2, q_3, q_4\} &) \\
(& D, & \neg\pi_1 \wedge \neg\pi_2 \wedge \pi_3, & \{q_3, q_5\} &) \\
(& D, & \neg\pi_1 \wedge \neg\pi_2 \wedge \neg\pi_3, & \emptyset &)
\end{array}$$

2.3.3. *Complementation*

If the determinizer also maintains the empty subset of states (cf. the last line in the previous example), then the resulting determinized automaton is *complete*: for each state a transition is applicable for each symbol of the alphabet. This property is important in order to define complementation. If an automaton M_1 with final states $F \subseteq Q$ is deterministic and complete, then an automaton accepting the language $\overline{L(M_1)}$ is obtained from M_1 simply by replacing F with $Q - F$.

As usual, the difference operation is defined straightforwardly in terms of complementation and intersection: if A and B are regular languages, then $A - B$ is defined as $A \wedge \overline{B}$.

2.3.4. *Minimization*

In Hopcroft's minimization algorithm (Hopcroft, 1971; Aho et al., 1974) a situation arises very similar to the determinization case. In this minimization algorithm, a partition of states is repeatedly refined by considering a pair of state and symbol which might reveal that an existing subset must be split. Rather than considering a pair of state and symbol, we consider in the generalization a pair of state and 'exclusive' predicate. As in the determinization algorithm we therefore need to consider all boolean combinations over the predicates present on a given state. In the actual implementation, we re-use the additional code required for the determinization algorithm in the implementation of the minimization algorithm.

The generalized minimization algorithm produces a pfsr that is minimal in the number of states. However, the pfsr is not necessarily unique, and could also be non-minimal in the number of transitions. This is caused by the fact that the predicates used in the pfsr might not be sufficiently general. For example, the language $\{a, b, c\}$ can be presented

[4] An implementation might choose to ignore transitions for which the corresponding predicate is not satisfiable.

with a 2-state automaton with a single transition labeled $\in \{a, b, c\}$, but e.g. also with a 2-state automaton with two transitions labeled respectively by $\in \{a, b\}$ and $\in \{c\}$. Therefore, the minimization of a pfsr includes a final *cleanup* step in which for each pair of states p and q all transitions from p to q with labels $\pi_1 \ldots \pi_i$ are combined into a single transition from p to q with associated label $\pi_1 \vee \ldots \vee \pi_i$. It turns out that in the case of transducers, the corresponding cleanup operator is more difficult, as we discuss in section 5.1.

3. Transducers with Predicates and Identities

3.1. Definitions

A predicate-augmented finite state transducer (pfst) M is a tuple $(Q, \Sigma, \Pi, E, S, F)$ with Q a finite set of states, Σ a set of symbols, Π a set of predicates over Σ. As before, S and F are sets of start states and final states respectively. E is a finite set $Q \times (\Pi \cup \{\epsilon\}) \times (\Pi \cup \{\epsilon\}) \times Q \times \{0, 1\}$. The final component of a transition is used to indicate identities. For all transitions $(p, d, r, q, 1)$ it must be the case that $d = r \neq \epsilon$.[5]

We define the function *str* from $\Pi \cup \{\epsilon\}$ to 2^{Σ^*}.

$$\begin{aligned} \mathrm{str}(\epsilon) &= \{\epsilon\} \\ \mathrm{str}(\pi) &= \{\sigma \in \Sigma | \pi(\sigma)\} \end{aligned}$$

If $\pi \in \Pi$ and $\mathrm{str}(\pi)$ is a singleton set, then the transitions (p, π, π, q, i) where $i \in \{0, 1\}$ are equivalent.

The relation $\widehat{E} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined inductively.

1. for all p: $(p, \epsilon, \epsilon, p) \in \widehat{E}$.

2. for all $(p, \phi, \psi, q, 0) \in E, x \in \mathrm{str}(\phi), y \in \mathrm{str}(\psi)$: $(p, x, y, q) \in \widehat{E}$.

3. for all $(p, \pi, \pi, q, 1) \in E$, $x \in \mathrm{str}(\pi)$: $(p, x, x, q) \in \widehat{E}$.

4. if (q_0, x_1, y_1, q_1) and (q_1, x_2, y_2, q) are both in $\widehat{E}$ then $(q_0, x_1x_2, y_1y_2, q) \in \widehat{E}$

The relation $R(M)$ accepted by a pfst M is defined to be $\{(w_d, w_r) | q_s \in S, q_f \in F, (q_s, w_d, w_r, q_f) \in \widehat{E}\}$.

[5] Note that without loss of generality we assume that there is no separate input and output alphabet, nor separate sets of predicates for input and output.

3.2. Operations on transducers

It is immediately clear that if Σ is finite, a pfst defines a regular relation. Therefore, the relations defined by a pfst are closed under various operations such as *union*, *concatenation*, *Kleene closure* and *composition*. From a practical point of view, it is important to note that it is possible to adapt the constructions for classical transducers for pfst.

The introduction of predicates over symbols is straightforward for operations such as *union*, *concatenation*, *Kleene closure* and *cross-product*. The *identity* and *composition* operations are described now as follows.

3.2.1. *Identity*

The identity relation for a given language L is $id(L) = \{(w, w)|w \in L\}$. For a given pfsr $M = (Q, \Sigma, \Pi, E, S, F)$, the identity relation is given by the pfst $M' = (Q, \Sigma, \Pi, E', S, F)$. Note that it would be wrong to define $E' = \{(p, \pi, \pi, q, 0)|(p, \pi, q) \in E\}$. Suppose π is true only of σ_1, σ_2. The pair $\pi : \pi$ then would be true of the pairs of symbols $\{(\sigma_1, \sigma_1), (\sigma_1, \sigma_2), (\sigma_2, \sigma_1), (\sigma_2, \sigma_2)\}$, whereas identity requires that we only allow the pairs $\{(\sigma_1, \sigma_1), (\sigma_2, \sigma_2)\}$. Another example to stress the point: the expression `identity(?)` ('copy') is quite different from `?:?` ('garbage-in garbage-out'). It is therefore necessary to introduce an identity marker for each of the transitions. The identity of a pfsr $M = (Q, \Sigma, \Pi, E, S, F)$ is given by $id(M) = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \pi, \pi, q, 1)|(p, \pi, q) \in E\}$.

The operations *domain*, *range* and *inverse* are straightforward. For a given pfst $M = (Q, \Sigma, \Pi, E, S, F)$, we have:

- domain($R(M)$) is given by the pfsr $M' = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \phi, q)|(p, \phi, \psi, q, i) \in E\}$.
- range($R(M)$) is given by the pfsr $M' = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \psi, q)|(p, \phi, \psi, q, i) \in E\}$.
- inverse($R(M)$) is given by the pfst $M' = (Q, \Sigma, \Pi, E', S, F)$ where $E' = \{(p, \psi, \phi, q, i)|(p, \phi, \psi, q, i) \in E\}$.

3.2.2. *Composition*

The composition of two binary relations is $R_1 \circ R_2 = \{(x_1, x_3)|(x_1, x_2) \in R_1, (x_2, x_3) \in R_2\}$. The composition operation is perhaps the most important operation on transducers. Its implementation is similar to the intersection operation for recognizers. In the classical case, a transducer for the composition of two given transducers M_1 and M_2 is constructed by considering the cross product of states of M_1 and M_2. A transition $((p_1, p_2), \sigma_d, \sigma_r, (q_1, q_2))$ exists iff there is some σ such that the corresponding transition $(p_1, \sigma_d, \sigma, q_1)$ exists in M_1 and $(p_2, \sigma, \sigma_r, q_2)$ exists in M_2. In the case of pfst a similar construction can be used, but instead of requiring that the output part of a transition in M_1 is identical to the input part of a transition in M_2, we now merely require that the conjunction of both predicates is satisfiable. In the case of identities, some further complications arise. The effect of combining two transitions is defined by means of the function *ct* that takes two transitions and returns a new transition:

$$\begin{array}{ll}
\mathrm{ct}((p_1, \pi_1, \pi_1, q_1, 1), (p_2, \pi_2, \pi_2, q_2, 1)) & = ((p_1, p_2), \pi_1 \wedge \pi_2, \pi_1 \wedge \pi_2, (q_1, q_2), 1) \\
\mathrm{ct}((p_1, \phi, \pi_1, q_1, 0), (p_2, \pi_2, \pi_2, q_2, 1)) & = ((p_1, p_2), \phi, \pi_1 \wedge \pi_2, (q_1, q_2), 0) \\
\mathrm{ct}((p_1, \pi_1, \pi_1, q_1, 1), (p_2, \pi_2, \psi, q_2, 0)) & = ((p_1, p_2), \pi_1 \wedge \pi_2, \psi, (q_1, q_2), 0) \\
\mathrm{ct}((p_1, \phi, \pi_1, q_1, 0), (p_2, \pi_2, \psi, q_2, 0)) & = ((p_1, p_2), \phi, \psi, (q_1, q_2), 0) \\
& \quad \text{if satisfiable}(\pi_1 \wedge \pi_2)
\end{array}$$

Note that this function is not defined in case either the input part of the second transition or the output part of the first transition is ϵ. These cases are treated separately in the definition below. Given two pfst $M_1 = (Q_1, \Sigma, \Pi, E_1, S_1, F_1)$ and $M_2 = (Q_2, \Sigma, \Pi, E_2, S_2, F_2)$, the relation $R(M_1) \circ R(M_2)$ is defined by $M = (Q_1 \times Q_2, \Sigma, \Pi, E, S_1 \times S_2, F_1 \times F_2)$ where

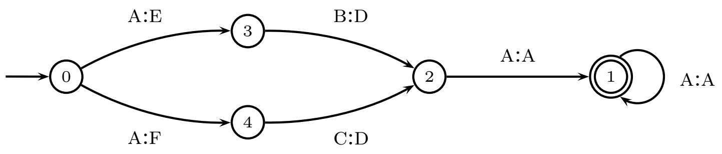
$$\begin{array}{rl}
E = & \{ct(e_1, e_2) | e_1 \in E_1, e_2 \in E_2\} \\
\cup & \{((p_1, p_2), \epsilon, \psi, (p_1, q_2), 0) | p_1 \in Q_1, (p_2, \epsilon, \psi, q_2, 0) \in E_2\} \\
\cup & \{((p_1, p_2), \phi, \epsilon, (q_1, p_2), 0) | p_2 \in Q_2, (p_1, \phi, \epsilon, q_1, 0) \in E_1\}
\end{array}$$

3.3. Determinization of Transducers

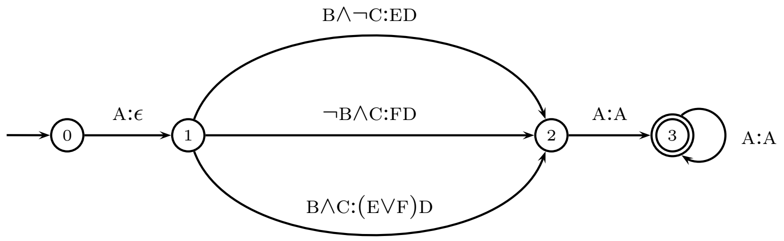
We will call a pfst M *deterministic* if M has a single start state, if there are no states $p, q \in Q$ such that $(p, \epsilon, \psi, q, i) \in E$, and if for all states p and symbols σ there is at most one transition (p, π_d, ψ, q, i) such that $\pi_d(\sigma)$. The transduction of an input string by means of a deterministic pfst is simple: in going through the input from left to right, you know exactly in which state you are (so there is no backtracking; alternatively if a parallel implementation is considered, there is no need to maintain a number of states linear in the size of the transducer). If a pfst M is deterministic then computing the transductions of a given string w as defined by M can be implemented efficiently. This computation is linear in w, and independent on the size of M. Since w can have several transductions (unless M is functional), we assume that this computation constructs a pfsr accepting $\{w' | (w, w') \in R(M)\}$.[6]

[6] As is well-known, not all finite-state transductions can be encoded by a deterministic transducer. As an example, a transduction which maps every `a` to a `b` if the

In order to extend the determinization algorithm for transducers (Oncina et al., 1993; Reutenauer, 1993; Mohri, 1996; Roche and Schabes, 1995; Roche and Schabes, 1997), we must extend pfst in such a way that the output part of a transition is a sequence of predicates. This extension is described later, but first we illustrate some of the complications that arise. For the moment we will simply assume that the output part of a transition contains a sequence of predicates. We first create an equivalent pfst which has no ϵ on the domain part of transitions, using the same technique as described in (Roche and Schabes, 1997, page 29).[7] In the determinization algorithm, local ambiguities such as those encountered in state 0 in (here, A... F are arbitrary predicates $\in \Pi$):



are solved by delaying the outputs as far as needed, until these symbols can be written out deterministically:[8]



The determinization algorithm for transducers maintains sets of pairs $Q \times \Pi^*$. Such a set corresponds to a state in the determinized transducer. In order to compute the transitions leaving such a set of pairs P, we compute for each π, $\mathit{Trans}^P(\pi) = \{(q, \phi\psi) | (p, \phi) \in P, (p, \pi, \psi, q) \in E\}$. In the example, we can be in states 3 and 4 after

input is of even length, and which maps every a to itself otherwise is a finite-state transduction, but cannot be encoded deterministically.

[7] We represent emissions associated with final states, as they surface in the determinization algorithm below, using an extra transition with ϵ as the domain part. We thus allow transitions (p, ϵ, ψ, q) only in case q is a final state and there are no transitions leaving q.

[8] By 'writing out deterministically' we mean writing out with a deterministic state transition. Such 'deterministic' outputs may still in the end be rejected if for some input, the machine ends in a non-final state.

reading a symbol compatible with A, with pending outputs E and F. We thus have $P = \{(3, \textsc{e}), (4, \textsc{f})\}$. Therefore, we have:

$$\mathit{Trans}^P(\textsc{b}) = \{(2, \textsc{ed})\},\ \mathit{Trans}^P(\textsc{c}) = \{(2, \textsc{fd})\}$$

Let Π' be the predicates in the domain of Trans^P. For each split of Π' into $\pi_1 \dots \pi_i$ and $\pi_{i+1} \dots \pi_n$ we have a proto-transition:

$$(P, \pi_1 \wedge \dots \wedge \pi_i \wedge \neg\pi_{i+1} \wedge \dots \wedge \neg\pi_n, \mathit{Trans}^P(\pi_1) \cup \dots \cup \mathit{Trans}^P(\pi_i))$$

In the example, we have the following proto-transitions (we need not represent the $\emptyset$ state):

$$\begin{array}{lllll} (& P, & \textsc{b} \wedge \neg\textsc{c}, & \{(2, \textsc{ed})\} &) \\ (& P, & \neg\textsc{b} \wedge \textsc{c}, & \{(2, \textsc{fd})\} &) \\ (& P, & \textsc{b} \wedge \textsc{c}, & \{(2, \textsc{ed}), (2, \textsc{fd})\} &) \end{array}$$

A transition is created from a proto-transition by removing the longest common prefix of predicates in the target pairs; this prefix is the sequence of output predicates of the resulting transition. However, before we remove this longest common prefix, we first consider possible simplifications in the sequences of output predicates, by packing multiple sequences associated with the same target state into a smaller number of sequences (using disjunction). In particular, two pairs of target state and predicate sequences (p_1, ψ_1) and (p_2, ψ_2) can be combined into a single pair (p, ψ) iff $p_1 = p_2 = p$ and $\psi_1 = \pi_1 \dots \pi_i \dots \pi_n$, $\psi_2 = \pi_1 \dots \pi'_i \dots \pi_n$ and $\psi = \pi_1 \dots \pi_i \vee \pi'_i \dots \pi_n$. In a proto-transition this simplification is applied repeatedly until no further simplifications are possible.

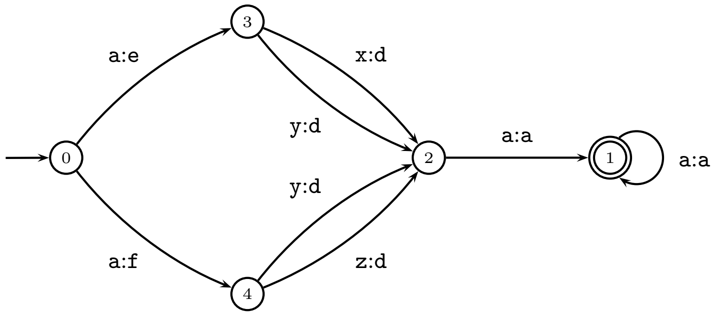
Here, the third proto-transition is simplified into:

$$(\ P,\ \textsc{b} \wedge \textsc{c},\ \{(2, (\textsc{e} \vee \textsc{f})\textsc{d})\}\ )$$

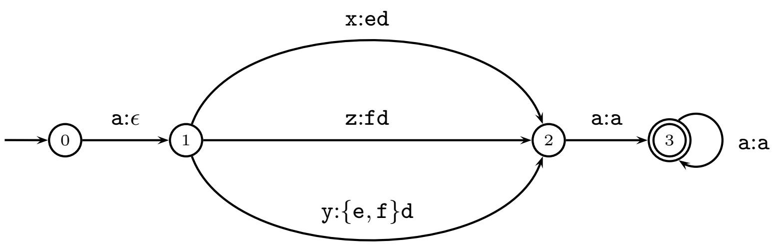
Moving the longest common prefix into the output part of the label yields:

$$\begin{array}{lllll} (& P, & \textsc{b} \wedge \neg\textsc{c} : \textsc{ed}, & \{(2, \epsilon)\} &) \\ (& P, & \neg\textsc{b} \wedge \textsc{c} : \textsc{fd}, & \{(2, \epsilon)\} &) \\ (& P, & \textsc{b} \wedge \textsc{c} : (\textsc{e} \vee \textsc{f})\textsc{d}, & \{(2, \epsilon)\} &) \end{array}$$

The introduction of predicates thus has the interesting effect that certain non-functional transducers can be treated by the transducer determinization algorithm. Assume that B is the predicate {`x,y`}, C is the predicate {`y,z`} and the predicates A, D, E and F are true only of the symbols `a`, `d`, `e` and `f` respectively. The equivalent normal transducer is:

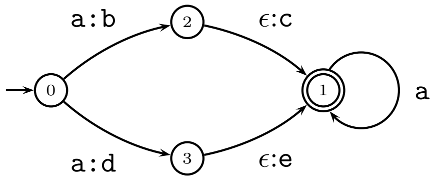


This transducer cannot be treated by the transducer determinization algorithm (that algorithm does allow a limited form of ambiguity, but only if this ambiguity can be delayed to a final state; here this is not possible). However, the same transduction can be determinized if expressed by a pfst:



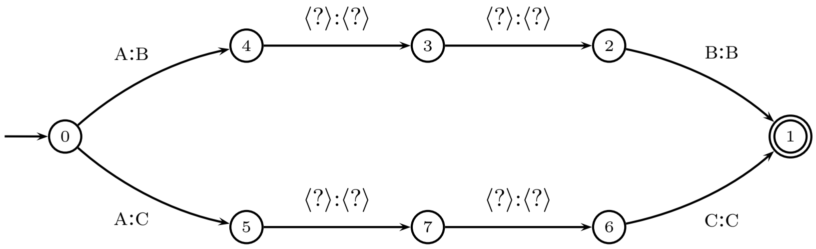
If predicates are used, then a larger class of transductions can be implemented efficiently. A precise classification of this class is beyond the scope of this paper, but note that the type of ambiguities that can be implemented in this way is limited to ambiguities that extend only over a single symbol.[9] For instance, a simple example such as the following cannot be determinized:

[9] Of related interest is the approach of (Kempe, 2000). He shows that ambiguous transductions can be computed efficiently by factorizing an ambiguous transducer T into a functional transducer T_1 and an ambiguous transducer T_2 such that T is equivalent to $T_1 \circ T_2$, and such that T_2 contains no 'failing paths'. In typical cases, T_1 contains meta-symbols which are expanded in T_2. This approach is more general in the sense that these meta-symbols range over sequences of symbols, rather than single symbols. It is more limited in the sense that identities cannot be expressed.

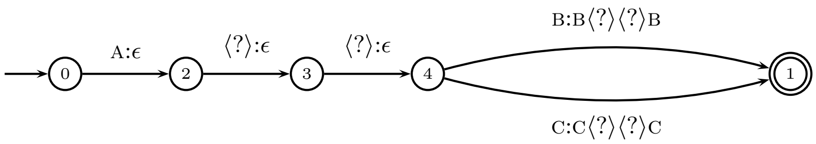


3.4. Determinization and identities

To treat identities, we must assume in the definition of proto-transition that if one of the positively occurring predicates in the boolean combination is associated with an identity, then the resulting predicate is associated with an identity as well. As an example consider the following transducer. For simplicity we assume B and C are mutually exclusive predicates; as before ? is a predicate which is true of all symbols. Also, we write $\langle$A$\rangle$:$\langle$A$\rangle$ for a transition A:A with an associated identity constraint.



Determinization produces:

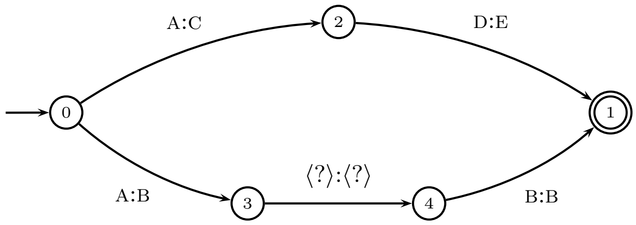


Outputs associated with an identity are delayed like ordinary outputs. Generalizing an idea due to Tamás Gaál and Lauri Karttunen[10] transducers with such disconnected identities are interpreted as follows. During the transduction of a string, a queue is maintained. Each time

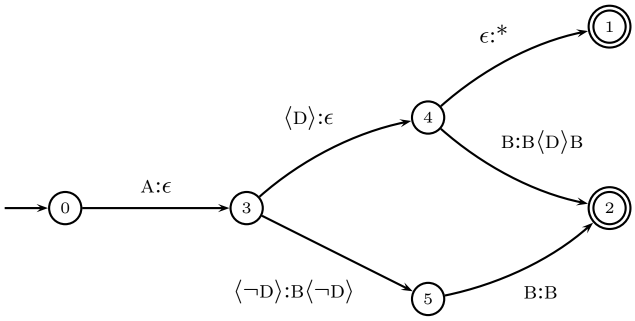
[10] personal communication

an input symbol is matched by a predicate with an associated identity, this symbol is enqueued. If a symbol matched by the corresponding predicate on the output side has to be written, then that symbol is obtained by a dequeue operation. With this use of a queue, our method for interpreting a transducer is no longer finite state. The transducer itself, however, still encodes a regular transduction.

A complication arises in cases like:



Determinization yields:



What sequence of output predicates should be put on the position of the *? According to the definitions, we get CE. However, this is not right because then there is a path $0 \rightarrow 3 \rightarrow 4 \rightarrow 1$ which has an identity on the input side without a corresponding identity on the output. Embedding such examples would lead to transducers in which identities are 'out of sync'. The determinization algorithm is therefore extended by marking in the output part that the scope of an identity ends; procedurally such a mark is interpreted as a dequeue operation which ignores the dequeued value. We write such a mark as $\langle\rangle$. In the example the sequence of outputs X becomes CE$\langle\rangle$. In the definition of proto-transition, if at least one of the positively occurring π_k has an associated identity then we append a $\langle\rangle$ mark to each of the outputs $\mathit{Trans}^P(\pi_l)$ for which π_l was not associated with an identity.

3.5. Finite State Transducers with a bounded Queue

We are now ready to define predicate-augmented finite state transducers with a bounded queue. A predicate-augmented finite state transducer with queue (qpfst) M is a tuple $(Q, \Sigma, \Pi, E, S, F)$ with Q a finite set of states, Σ a set of symbols, Π a set of predicates over Σ. As before, S and F are sets of start states and final states respectively. E is a finite set $Q \times ((\Pi \cup \{\epsilon\}) \times \{0,1\}) \times ((\Pi \cup \{\lambda\}) \times \{0,1\})^* \times Q$.

In a transition, each predicate is associated with a queue marker, which is one of $\{0,1\}$. On the input side, 1 will imply an enqueue operation of the symbol matching the predicate; on the output side 1 will imply a dequeue operation of the symbol matching the predicate. In the input part of the transition, ϵ can be used as well, in which case the queue marker must be 0 (input epsilons will be employed to represent outputs associated with initial and final states). In the output part of a transition we can have λ instead of a predicate, in order to represent the explicit dequeue operations motivated earlier. We require that every λ must have a corresponding queue marker which is 1.

The relation $O : ((\Pi \cup \{\lambda\}) \times \{0,1\})^* \times \Sigma^* \times \Sigma^* \times \Sigma^*$ determines the effect of the output part of a transition. Its arguments represent respectively the output sequence of a transition, the (incoming and outgoing) queues, and the resulting output string. Note that queues are written from left to right in such a way that an element is enqueued to the left and dequeued from the right.

1. for all $x \in \Sigma^*$ we have $(\epsilon, x, x, \epsilon) \in O$
2. if $(\phi, x_0, x, z) \in O$ then for all $\sigma \in \Sigma$ we have $((\lambda, 1)\phi, x_0\sigma, x, z) \in O$
3. if $(\phi, x_0, x, z) \in O$ then for all $\sigma \in \Sigma$ and $\pi \in \Pi$ such that $\pi(\sigma)$ we have $((\pi, 1)\phi, x_0\sigma, x, \sigma z) \in O$
4. if $(\phi, x_0, x, z) \in O$ then for all $\sigma \in \Sigma$ and $\pi \in \Pi$ such that $\pi(\sigma)$ we have $((\pi, 0)\phi, x_0, x, \sigma z) \in O$

The relation $\widehat{E} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q \times \Sigma^* \times \Sigma^*$ is a relation between source states, sequences of input symbols, sequences of output symbols, target states, and source- and target queues. It is defined inductively.

1. for all $p \in Q$, $(p, \epsilon, \epsilon, p, \epsilon, \epsilon) \in \widehat{E}$.
2. for each transition $(p, (\epsilon, 0), \phi, q) \in E$ such that $(\phi, x_0, x, w) \in O$, $(p, \epsilon, w, q, x_0, x) \in \widehat{E}$
3. for each transition $(p, (\pi, 0), \phi, q) \in E$ such that $\pi(\sigma)$ and $(\phi, x_0, x, w) \in O$, $(p, \sigma, w, q, x_0, x) \in \widehat{E}$.

4. for each transition $(p, (\pi, 1), \phi, q) \in E$ such that $\pi(\sigma)$ and $(\phi, \sigma x_0, x, w) \in O$, $(p, \sigma, w, q, x_0, x) \in \widehat{E}$.

5. if $(q_0, x_1, y_1, q_1, x_0, x_1)$ and $(q_1, x_2, y_2, q, x_1, x)$ are both in $\widehat{E}$ then $(q_0, x_1 x_2, y_1 y_2, q, x_0, x) \in \widehat{E}$

The relation $R(M)$ accepted by a qpfst M is defined to be $\{(w_d, w_r) | q_s \in S, q_f \in F, (q_s, w_d, w_r, q_f, \epsilon, \epsilon) \in \widehat{E}\}$.

Such qpfst are generally very powerful. However, the qpfst which result from the generalized transducer determinization algorithm are all limited. Not only are these transducers deterministic by construction, but they are also limited in the way the queue is actually used: in each case the maximum size of the queue is some constant. And of course, since the input was a finite-state transducer, the resulting equivalent qpfst describes a finite-state transduction too. Another way to characterize this limited use of qpfst is to observe that in such cases every cyclic path through such a transducer will have identical input and output queue: the queue is only used in a strictly local sense.

The ordinary transducer determinization algorithm is guaranteed to terminate only if the input transducer can be determined, i.e., the transducer must be *sub-sequential*. A separate algorithm exists to check a given transducer for subsequentiality (section 5.2). The same termination property holds for the generalized transducer determinization algorithm. If the generalized transducer determinization algorithm terminates for a given pfst, then the result is an equivalent deterministic qpfst. The application of a determinized (potentially non-functional) qpfst T to a given string w is linear in the size of w, and independent of the size of T.

3.6. Synchronization

Operations such as composition are defined for pfst. Therefore, we have implemented an operator which transforms a given bounded qpfst back into pfst by synchronizing the identities. Of course, the resulting pfst will generally not be deterministic anymore.

The synchronization is implemented by an algorithm which maintains an agenda of 'synchronous states' (initialized by the set of start states). For each state on the agenda minimal synchronous paths are generated. The target states of these paths are added to the agenda, and these paths themselves are broken into pieces such that each piece is synchronous (by introducing transitions with ϵ on the input or output side).

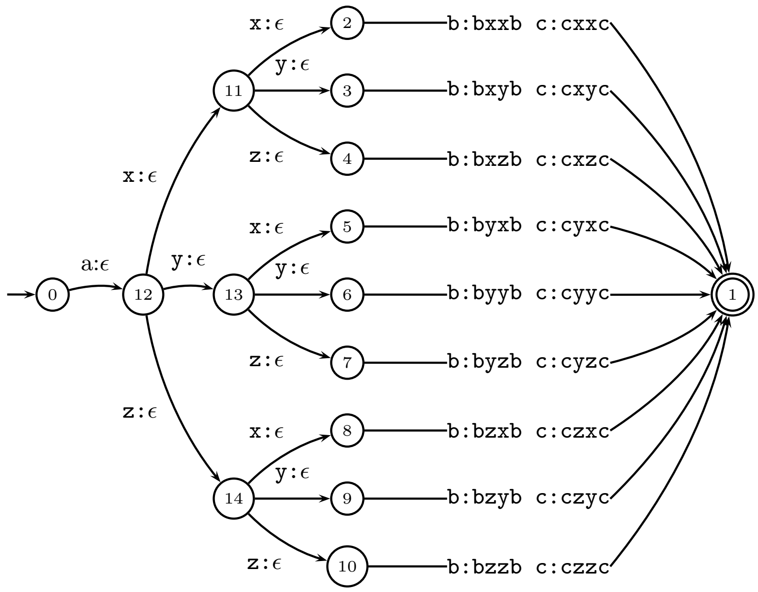
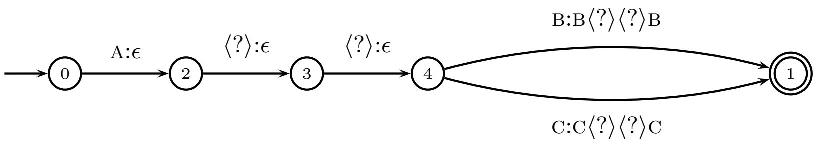


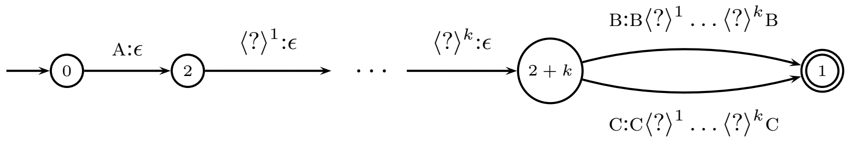
Figure 1. A minimal subsequential transducer without predicates. The equivalent minimal transducer employing predicates only has 5 states and 5 transitions.

3.7. Succintness

Predicate-augmented finite state transducers typically require fewer transitions than classical finite state transducers, by an argument similar to that for pfsr. In the case of predicate-augmented pfst with bounded queue, however, the number of states can often be much smaller than the number of states in an equivalent, classical, subsequential transducer. Consider again the first example in section 3.4. Application of our variant of Mohri's determinization algorithm yields a transducer of 5 states and 5 transitions, repeated here for convenience:



Suppose we were to expand this example into a classical subsequential transducer, then depending on the size of the alphabet, the resulting transducers would have many more states. The example for the alphabet {x,y,z} with 15 states and 31 transitions is given in figure 1; for an alphabet of 26 symbols, the result already has 705 states and 2055 transitions. For an alphabet of 254 symbols, the result has 64773 states and 193803 transitions. Instead of having two question marks on the input side in a row, consider similar examples where we have k such question marks in a row:



In these cases, the minimal qpfst will have $3+k$ states. The equivalent minimal subsequential transducer will require $3 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^k$ states. An analysis of the difference in succintness in terms of descriptional complexity (e.g. (Dassow et al., 1997)) is beyond the scope of this article; but this class of examples suggests that there are arbitrarily many relations for which the qpfst device requires exponentially fewer states than subsequential transducers.

4. Practical Considerations

Predicate-augmented finite state automata are fully integrated in version 6 of the Fsa Utilities toolbox. The toolbox is freely available from `http://www.let.rug.nl/~vannoord/Fsa/`. In addition, some of the algorithms have been implemented in `C++`.

4.1. Membership and Non-membership Predicates

In practice, we have mostly assumed that all predicates are of the form $\in S$ and $\notin S$ for arbitrary finite sets of symbols S. The non-membership predicates are very useful to specify in a compact form large (potentially infinite) sets of symbols. A boolean combination of membership and non-membership predicates can always be written in this form, as the following table shows:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$
$\in S_1$	$\in S_2$	$\notin S_1$	$\in S_1 \cap S_2$	$\in S_1 \cup S_2$
$\in S_1$	$\notin S_2$		$\in S_1 - S_2$	$\notin S_2 - S_1$
$\notin S_1$	$\in S_2$	$\in S_1$	$\in S_2 - S_1$	$\notin S_1 - S_2$
$\notin S_1$	$\notin S_2$		$\notin S_2 \cup S_1$	$\notin S_1 \cap S_2$

In the implementation, any boolean combination of predicates that occurs is immediately rewritten into this atomic form. Determining whether a symbol satisfies a predicate is trivial. Determining satisfiability of an atomic formula is trivial too: the only atomic formula that is not satisfiable is $\in \emptyset$. The actual computation thus involves standard operations on sets: membership, union, intersection and difference. The implementation provides three alternative implementations, by representing sets as ordered lists, bit vectors or balanced binary trees.

The system also supports the addition of various application-specific predicate sets. There are various possibilities here. For instance, predicates could be expressed in terms of type hierarchies as in (Carpenter, 1992). Another possibility is a predicate module in which predicates are membership tests of regular languages. A syntax component could be implemented by a pfsr in which predicates describe words. These predicates themselves might be implemented by finite automata over character strings. If predicates get complicated, the efficiency of checking such predicates may become important.

4.2. Smaller Transducers

The operations on predicate-augmented finite state recognizers and transducers discussed here have been fully implemented and integrated in a finite state toolkit. Although the implementation of these operations is more involved than for normal automata it turned out that the introduction of predicates has improved performance considerably, because automata are smaller.

For example, consider the soundex algorithm expressed as a regular expression, presented at the Xerox web-site.[11] The soundex algorithm maps proper names to four-letter codes, where 'similar' names are assigned the same code. This algorithm can be used to match names that are misspelled, for instance due to poor handwriting or voice transmission; similar problems occur in historical archives. A description of the algorithm and some historical remarks are given in (Knuth, 1998). The compilation of the soundex regular expression yields a transducer with 1217 transitions. By design, the soundex algorithm treats various classes of characters identically. Using predicates for each of these classes yields a transducer with 198 transitions. The construction is four times faster as well. Depending on how predicates are implemented, running the resulting transducer might be slower. In our experiments these effects were not noticeable.

[11] `http://www.rxrc.xerox.com/research/mltt/fst/fsexamples.html`

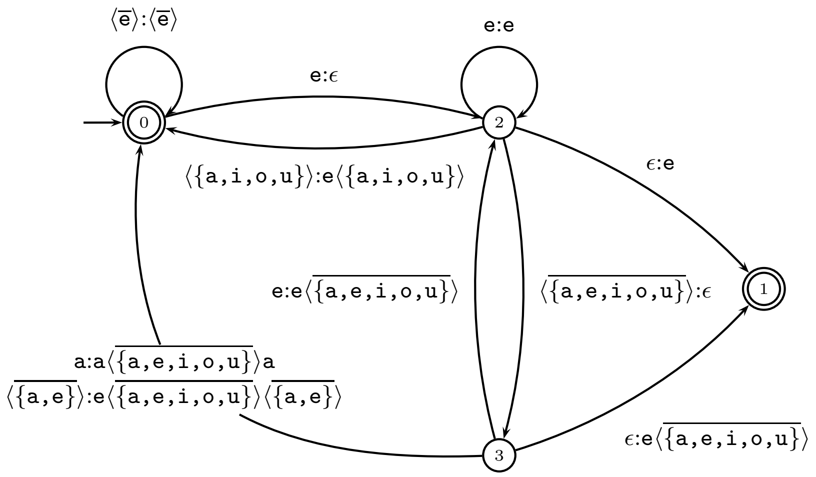


Figure 2. A minimal transducer with predicates implementing the phonological rule `e` $\rightarrow$ `a/__C a`. The equivalent minimal transducer without predicates has 24 states and 620 transitions.

The observation that the use of predicates generally leads to transducers with fewer states can be observed in practically relevant examples as well. Consider the following hypothetical phonological rule:

$$\texttt{e} \rightarrow \texttt{a}/\_\texttt{C a}$$

This rule indicates that an `e` should be mapped to an `a` if it is followed by a consonant and an `a`. Assuming an alphabet consisting of 5 vowels and 21 consonants, the corresponding minimal transducer for this example consists of 24 states and 620 transitions. If predicates are used, the resulting automaton only has 4 states and 10 transitions (cf. figure 2).

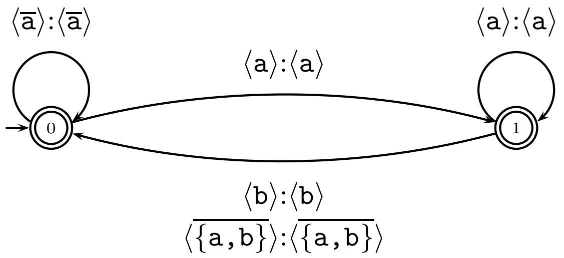
5. Future Work

5.1. Minimization

The minimization algorithm for transducers (Mohri, 1994; Mohri, 2000) can be applied to a bounded qpfst without modifications. The transducer minimization algorithm consists of two steps. In the first step, all output symbols are moved into preceding transitions as much as is possible. This is done by computing for each state the longest common prefix of the outputs associated with all paths from that state to a final state. The second step of the transducer minimization algorithm consists of the application of ordinary recognizer minimization to the resulting transducer, temporarily treating the labels as atomic symbols.

The application of the transducer minimization algorithm to a bounded qfst might result in a qpfst with identities in which the output has to be produced before the corresponding input symbol has been observed. The queue-mechanism can be generalized to treat such cases as well. We use an implementation of queues described in (Sterling and Shapiro, 1994, page 299) in which an element can be dequeued before it is enqueued. The output is a variable temporarily; obviously this requires output to be buffered. We implemented this both in `C++` as well as in Prolog.

However, applying the transducer minimization algorithm in this way does not neccessarily produce a minimal qpfst. One problem is that in the transducer minimization algorithm, the final step consists of an application of the recognizer minimization algorithm in such a way that the labels of the transducer are temporarily treated as unanalyzable atoms. This works in the case of ordinary transducers, but is not good enough for our purposes. The following example illustrates this particular problem.



The transduction implemented by this transducer is simply the identity relation over Σ^*. However, the application of the transducer minimization algorithm will produce an identical transducer, rather than the minimal one.

In the implementation in the Fsa Utilities we have constructed a variety of heuristics, which includes a generalization of the transducer minimization algorithm, in order to reduce the size of deterministic transducers. In most practical cases, the heuristics produce a minimal transducer.

5.2. Subsequentiality and Bi-machines

Recall that the transducer determinization algorithm is guaranteed to terminate only in case the input transducer can be determinized, i.e., the transducer describes a subsequential transduction. Therefore, it is important to implement an algorithm which checks for this property. We are working on an algorithm to check subsequentiality of a given pfst, based on the algorithm presented in (Roche and Schabes, 1997). We have adapted the algorithm proposed in (Roche and Schabes, 1997) since it fails to treat certain types of transducer correctly; we intend to provide details somewhere else.

A further natural extension is the generalization of bi-machines and the related algorithms to the case of predicates.

Acknowledgements

This research was partly carried out within the framework of the PIONIER Project *Algorithms for Linguistic Processing*, funded by NWO (Dutch Organization for Scientific Research) and the University of Groningen. We are grateful to Gosse Bouma, Jan Daciuk, Rob Malouf, Mark-Jan Nederhof, Bruce Watson, Franck Thollard and Markus Walther for comments.

References

Aho, A. V., J. E. Hopcroft, and J. D. Ullman: 1974, *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A. V., R. Sethi, and J. D. Ullman: 1986, *Compilers. Principles, Techniques and Tools*. Addison Wesley.

Bird, S. and T. M. Ellison: 1994, 'One-Level Phonology: Autosegmental Representations and Rules as Finite Automata'. *Computational Linguistics* **20**(1), 55–90.

Carpenter, B.: 1992, *The Logic of Typed Feature Structures*. Cambridge University Press, New York.

Daciuk, J.: 1998, 'Incremental Construction of Finite-state Automata and Transducers, and their Use in the Natural Language Processing'. Ph.D. thesis, Technical University of Gdańsk.

Daciuk, J.: 2000, 'Experiments with Automata Compression'. In: M. Daley, M. G. Eramian, and S. Yu (eds.): *Conference on Implementation and Application of Automata CIAA'2000*. London, Ontario, Canada, pp. 113–119, University of Western Ontario.

Dassow, J., G. Paun, and A. Salomaa: 1997, 'Grammars with Controlled Derivations'. In: G. Rozenberg and A. Salomaa (eds.): *Handbook of Formal Languages Vol.2 Linear Modeling: Background and Application*. Springer, pp. 101–154.

Eisner, J.: 1997, 'Efficient Generation in Primitive Optimality Theory'. In: *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*. pp. 313–320.

Gerdemann, D. and G. van Noord: 1999a, 'Transducers from Rewrite Rules with Backreferences'. In: *Ninth Conference of the European Chapter of the Association for Computational Linguistics*. Bergen Norway.

Gerdemann, D. and G. van Noord: 1999b, 'Transducers from Rewrite Rules with Backreferences'. In: *Ninth Conference of the European Chapter of the Association for Computational Linguistics*. Bergen Norway, pp. 126–133.

Gildea, D. and D. Jurafsky: 1996, 'Learning Bias and Phonological-Rule Induction'. *Computational Linguistics* **22**(4), 497–530.

Hopcroft, J. E.: 1971, 'An n log n algorithm for minimizing the states in a finite automaton'. In: Z. Kohavi (ed.): *The Theory of Machines and Computations*. Academic Press, pp. 189–196.

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

Johnson, J. H. and D. Wood: 1997, 'Instruction Computation in Subset Construction'. In: D. Raymond, D. Wood, and S. Yu (eds.): *Automata Implementation*. Springer Verlag, pp. 64–71. Lecture Notes in Computer Science 1260.

Karttunen, L.: 1991, 'Finite-state Constraints'. In: *Proceedings International Conference on Current Issues in Computational Linguistics*. Universiti Sains Malaysia, Penang, pp. 23–40.

Karttunen, L.: 1995, 'The Replace Operator'. In: *33th Annual Meeting of the Association for Computational Linguistics*. M.I.T. Cambridge Mass., pp. 16–23.

Karttunen, L.: 1996, 'Directed Replacement'. In: *34th Annual Meeting of the Association for Computational Linguistics*. Santa Cruz, pp. 108–115.

Karttunen, L., J.-P. Chanod, G. Grefenstette, and A. Schiller: 1996, 'Regular Expressions for Language Engineering'. *Natural Language Engineering* **2**(4), 305–238. http://www.rxrc.xerox.com/research/mltt/fst/articles/jnle-97/rele.html.

Kempe, A.: 2000, 'Factorization of Ambiguous Finite-State Transducers'. In: *CIAA 2000. Fifth International Conference on Implementation and Application of Automata. Preproceedings*. London, Ontario, Canada, pp. 157–164.

Kempe, A. and L. Karttunen: 1996, 'Parallel Replacement in the Finite-State Calculus'. In: *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*. Copenhagen, Denmark, pp. 622–627.

Kiraz, G. A.: 1999, 'Compressed Storage of Sparse Finite-State Transducers'. In: O. Boldt, H. Jürgensen, and L. Robbins (eds.): *Workshop on Implementing Automata WIA99 - Pre-Proceedings*. Potsdam.

Klarlund, N.: 1998, 'Mona & Fido: The Logic-Automaton Connection in Practice'. In: *Computer Science Logic, CSL '97*. LNCS 1414.

Knuth, D. E.: 1998, *The Art of Computer Programming, Volume 3, Sorting and Searching*. Addison Wesley, second edition edition.

Kowaltowski, T., C. L. Lucchesi, and J. Stolfi: 1993, 'Minimization of Binary Automata'. In: *First South American String Processing Workshop*. Belo Horizonte, Brasil.

Mohri, M.: 1994, 'Compact Representations by Finite-State Transducers'. In: *32th Annual Meeting of the Association for Computational Linguistics*. New Mexico State University, pp. 204–209.

Mohri, M.: 1996, 'On some applications of finite-state automata theory to natural language processing'. *Natural Language Engineering* **2**, 61–80. Originally appeared in 1994 as Technical Report, institut Gaspard Monge, Paris.

Mohri, M.: 2000, 'Minimization Algorithms for Sequential Transducers'. *Theoretical Computer Science* **234**, 177–201.

Oncina, J., P. García, and E. Vidal: 1993, 'Learning subsequential transducers for pattern recognition interpretation tasks'. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **15**, 448–458.

Perrin, D.: 1990, 'Finite Automata'. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*. Elsevier and the MIT Press, pp. 1–57.

Reutenauer, C.: 1993, 'Subsequential Functions: Characterizations, Minimization, Examples'. In: *Proceedings of the International Meeting of Young Computer Scientists*. Berlin, Springer. Lecture Notes in Computer Science.

Roche, E. and Y. Schabes: 1995, 'Deterministic Part-of-speech Tagging with Finite-state Transducers'. *Computational Linguistics* **21**(2), 227–253.

Roche, E. and Y. Schabes: 1997, 'Introduction'. In: E. Roche and Y. Schabes (eds.): *Finite-State Language Processing*. Cambridge, Mass: MIT Press.

Sterling, L. and E. Shapiro: 1994, *The Art of Prolog*. Cambridge Mass.: MIT Press. Second Edition.

van Noord, G. and D. Gerdemann: 1999, 'An Extendible Regular Expression Compiler for Finite-state Approaches in Natural Language Processing'. In: O. Boldt, H. Juergensen, and L. Robbins (eds.): *Workshop on Implementing Automata; WIA99 Pre-Proceedings*. Potsdam Germany.

Walther, M.: 1999, 'One-Level Prosodic Morphology'. Technical Report 1, Institüt für Germanistische Sprachwissenschaft, Philipps-Universität Marbug. cs.CL/9911011.

Walther, M.: 2000, 'Finite-State Reduplication in One-Level Prosodic Morphology'. In: *First Conference of the North American Chapter of the Association for Computational Linguistics*. Seattle, pp. 296–302.

Watson, B. W.: 1999a, 'Implementing and Using Finite Automata Toolkits'. In: A. Kornai (ed.): *Extended Finite State Models of Language*. Cambridge University Press, pp. 19–36.

Watson, B. W.: 1999b, 'The OpenFIRE Initiative'. In: J. Aoe (ed.): *Proceedings of the International Conference on Computer Processing of Oriental Languages*. Tokushima, Japan, pp. 421–424.