Parallel Replacement in Finite State Calculus

André Kempe and Lauri Karttunen
Rank Xerox Research Centre - Grenoble Laboratory
6, chemin de Maupertuis - 38240 Meylan - France
{kempe,karttunen}@xerox.fr http://www.xerox.fr/grenoble/mltt

Abstract

This paper extends the calculus of regular expressions with new types of replacement expressions that enhance the expressiveness of the simple replace operator defined in Karttunen (1995). Parallel replacement allows multiple replacements to apply simultaneously to the same input without interfering with each other. We also allow a replacement to be constrained by any number of alternative contexts. With these enhancements, the general replacement expressions are more versatile than two-level rules for the description of complex morphological alternations.

1 Introduction

A replacement expression specifies that a given symbol or a sequence of symbols should be replaced by another one in a certain context or contexts. Phonological rewrite-rules (Kaplan and Kay, 1994), two-level rules (Koskenniemi 1983), syntactic disambiguation rules (Karlsson et al 1994, Koskenniemi, Tapanainen, and Voutilainen 1992), and part-of-speech assignment rules (Brill 1992, Roche and Schabes 1995) are examples of replacement in context of finite-state grammars. Kaplan and Kay (1994) describe a general method representing a replacement procedure as finite-state transduction. Karttunen (1995) takes a somewhat simpler approach by introducing to the calculus of regular expression a replacement operator that is defined just in terms of the other regular expression operators. We follow here the latter approach.

In the regular expression calculus, the replacement operator, \(\rightarrow\), is similar to crossproduct, in that a replacement expression describes a relation between two simple regular languages. Consequently, regular expressions can be conveniently combined with other kinds of operations, such as composition and union to form complex expressions.

A replacement relation consists of pairs of strings that are related to one another in the manner sketched below:

\[
\begin{align*}
&x \quad u^1_i \quad y \quad u^2_i \quad z \quad \text{upper string} \quad [1] \\
&x \quad h^1_i \quad y \quad h^2_i \quad z \quad \text{lower string}
\end{align*}
\]

We use \(u^1_i\) and \(u^2_i\) to represent instances of \(U_i\) (with \(i \in [1, n]\)) and \(h^1_i\) and \(h^2_i\) to represent instances of \(L_i\). The upper string contains zero or more instances of \(U_i\), possibly interspersed with other material (denoted here by \(x\), \(y\), and \(z\)). In the corresponding lower string the sections corresponding to \(U_i\) are instances of \(L_i\), and the intervening material remains the same (Karttunen, 1995).

The \(\rightarrow\) operator makes the replacement obligatory, \((\rightarrow)\) makes it optional. For the sake of completeness, we also define the inverse operators, \(<\sim\) and \((\sim)\), and the bidirectional variants, \(\sim\) and \((\sim)\).

We have incorporated the new replacement expressions into our implementation of the finite-state calculus (Kempe and Karttunen, 1995). Thus, we can construct transducers directly from replacement expressions as part of the general calculus, without invoking any special rule compiler.

1.1 Simple regular expressions

The table below describes the types of regular expressions and special symbols that are used to define the replacement operators.

| Symbol | Description |
|--------|-------------|
| \(h\)  | option, \(A \mid 0\) |
| \(h^*\) | Kleene star |
| \(h^+\) | Kleene plus |
| \(h/B\) | ignore (A possibly interspersed with strings from B) |
| \(\sim\) | complement (negation) |
| \(\mathcal{A}\) | contains (at least one) A |
| \(A \cdot B\) | concatenation |
| \(A \cup B\) | union |
| \(A \cap B\) | intersection |
| \(A \ominus B\) | relative complement (minus) |
| \(A .x. B\) | crossproduct (Cartesian product) |
| \(A .o. B\) | composition |
| \(\epsilon\) | epsilon (the empty string) |
| \(\cdot\) | affects empty string replacement (sec. 2.2) |
| \(?\) | any symbol |
| \(?\) | the universal ("sigma-star") language (contains all possible strings of any length including the empty string) |
| \(\#\) | string beginning or end (sec. 2.1) |

Note that expressions that contain the cross-product \((.x.)\) or the composition \((.o.)\) operator, describe regular relations rather than regular languages. A regular relation is a mapping from one regular language to another one. Regular languages correspond to simple finite-state automata; regular relations are modelled by finite-state transducers.

In the relation \(A .x. B\), we call the first member, \(A\), the upper language and the second member, \(B\), the lower language. This choice of words is motivated by the linguistic tradition of writing the result of a rule application underneath the original form. In a cascade of compositions, \(R_1 .o. R_2 .o. \ldots .o. R_n\), which models a linguistic derivation by rewrite-rules, the upper side of the first relation, \(R_1\), contains the "underlying lexical
form", while the lower side of the last relation, \( R_n \), contains the resulting "surface form".

We recognize two kinds of symbols: simple symbols (a, b, c, etc.) and fst pairs \( (a:b, y:z, \text{etc.}) \). An fst pair \( a:b \) can be thought of as the cross product of a and b, the minimal relation consisting of a (the upper symbol) and b (the lower symbol).

2 Parallel Replacement

Conditional parallel replacement denotes a relation which maps a set of \( n \) expressions \( U_i \) (\( i \in [1, n] \)) in the upper language into a set of corresponding \( n \) expressions \( L_i \) in the lower language if, and only if, they occur between a left and a right context \( (l_i, r_i) \).

\[
\{ U_1 \rightarrow L_1 | l_1 \ldots r_1 \} , \ldots , \{ U_n \rightarrow L_n | l_n \ldots r_n \} \quad [3]
\]

Unconditional parallel replacement denotes a similar relation where the replacement is not constraint by contexts.

Conditional parallel replacement corresponds to what Kaplan and Kay (1994) call "batch rules" where a set of rules (replacements) is collected together in a batch and performed in parallel, at the same time, in a way that all of them work on the same input, i.e. not one applies to the output of another replacement.

2.1 Examples

Regular expressions based on [3] can be abbreviated if some of the upper-lower pairs, and/or some of the left-right pairs, are equivalent. The complex expression:

\[
\{ a \rightarrow b , b \rightarrow c | l x \ldots y \} ; \quad [4]
\]

which contains multiple replacement in one left and right context, can be written in a more elementary way as two parallel replacements:

\[
\{ a \rightarrow b | l x \ldots y \}, \{ b \rightarrow c | l x \ldots y \} ; \quad [5]
\]

Figure 1: Transducer encoding [4] and [5]. (Every arc with more than one label actually stands for a set of arcs with one label each.)

Figure 1 shows the state diagram of a transducer resulting from [4] or [5]. The transducer maps the string \( xaxayby \) to \( xaxbyby \) following the path \( 0 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \).

The complex expression:

\[
\{ a \rightarrow b , b \rightarrow c | l x \ldots y , v \ldots w \} , \quad [6]
\]

contains five single parallel replacements:

\[
\{ a \rightarrow b | l x \ldots y \} , \quad [7]
\]

\[
\{ a \rightarrow b | l v \ldots w \} , \quad [7]
\]

\[
\{ b \rightarrow c | l x \ldots y \} , \quad [7]
\]

\[
\{ b \rightarrow c | l v \ldots w \} , \quad [7]
\]

\[
\{ a \rightarrow c | l p \ldots q \} ;
\]

Contexts can be unspecified as in

\[
\{ a \rightarrow b | l x \ldots y , v \ldots w \} ; \quad [8]
\]

where a is replaced by b only when occurring between x and y, or after v, or before w.

An unspecified context is equivalent to \( \ast \ast \), the universal (sigma-star) language. Similarly, a specified context, such as \( x \ldots y \), is actually interpreted as \( x_\ast \ldots y_\ast \), that is, implicitly extending the context to infinity on both sides of the replacement.

This is a useful convention, but we also need to be able to refer explicitly to the beginning or the end of a string. For this purpose, we introduce a special symbol, \( \# \). (Kaplan and Kay, 1994, p. 349).

In the example

\[
\{ a \rightarrow b | l x \ldots ; , v \ldots ; , w \ldots ; \} ; \quad [9]
\]

a is replaced by b only when it is at the beginning of a string or between v and the two final symbols of a string.

2.2 Replacement of the Empty String

The language described by the upper part of a replacement expression can contain the empty string \( \varepsilon \). In this case, every string that is in the upper-side language of the relation, is mapped to an infinite set of strings in the lower-side language as the upper-side string can be considered as a concatenation of empty and non-empty substrings, with \( \varepsilon \) at any position and in any number. E.g.

\[
a^* \rightarrow x | l \ldots ; \quad [11]
\]

maps the string \( bb \) to the infinite set of strings \( bb, xbb, xbbx, xbbx, \text{etc.} \), since the language described by \( a^* \) contains \( \varepsilon \), and the string \( bb \) can be considered as a result of any one of the concatenations \( b \ast b \ast b \), \( \varepsilon \ast b \ast b \), \( c \ast b \ast b \), \( \varepsilon \ast c \ast b \ast b \), etc.

For many practical purposes it is convenient to construct a version of empty-string replacement that allows only one application between any two adjacent symbols (Karttunen, 1995). In order not to confuse the notation by a non-standard interpretation of the notion of empty string, we introduce a special pair of brackets, \( [\ldots \ldots] \), placed around the

\[
\{ a \rightarrow b | l x \ldots y , v \ldots w \} ; \quad [8]
\]

where a is replaced by b only when occurring between x and y, or after v, or before w.

An unspecified context is equivalent to \( \ast \ast \), the universal (sigma-star) language. Similarly, a specified context, such as \( x \ldots y \), is actually interpreted as \( x_\ast \ldots y_\ast \), that is, implicitly extending the context to infinity on both sides of the replacement.

This is a useful convention, but we also need to be able to refer explicitly to the beginning or the end of a string. For this purpose, we introduce a special symbol, \( \# \). (Kaplan and Kay, 1994, p. 349).

In the example

\[
\{ a \rightarrow b | l x \ldots ; , v \ldots ; , w \ldots ; \} ; \quad [9]
\]

a is replaced by b only when it is at the beginning of a string or between v and the two final symbols of a string.

2.2 Replacement of the Empty String

The language described by the upper part of a replacement expression can contain the empty string \( \varepsilon \). In this case, every string that is in the upper-side language of the relation, is mapped to an infinite set of strings in the lower-side language as the upper-side string can be considered as a concatenation of empty and non-empty substrings, with \( \varepsilon \) at any position and in any number. E.g.

\[
a^* \rightarrow x | l \ldots ; \quad [11]
\]

maps the string \( bb \) to the infinite set of strings \( bb, xbb, xbbx, xbbx, \text{etc.} \), since the language described by \( a^* \) contains \( \varepsilon \), and the string \( bb \) can be considered as a result of any one of the concatenations \( b \ast b \ast b \), \( \varepsilon \ast b \ast b \), \( c \ast b \ast b \), \( \varepsilon \ast c \ast b \ast b \), etc.

For many practical purposes it is convenient to construct a version of empty-string replacement that allows only one application between any two adjacent symbols (Karttunen, 1995). In order not to confuse the notation by a non-standard interpretation of the notion of empty string, we introduce a special pair of brackets, \( [\ldots \ldots] \), placed around the

\[
\{ a \rightarrow b | l x \ldots y , v \ldots w \} ; \quad [8]
\]
upper side of a replacement expression that presupposes a strict alternation of empty substrings and non-empty substrings of exactly one symbol:

$$e^x e^y e^z e^c \ldots$$  \[12\]

In applying this to the above example, we obtain

$$[.a*.] \rightarrow x \mid \ldots$$  \[13\]

that maps the string $bb$ only to $xbbbx$ since $bb$ is here considered exclusively as a result of the concatenation $e^c b^e c^e b^e \epsilon$.

If contexts are specified (in opposition to the above example) then they are taken into account.

### 2.3 The Algorithm

#### 2.3.1 Auxiliary Brackets

The replacement of one substring by another one inside a context, requires the introduction of auxiliary symbols (e.g. brackets). Kaplan and Kay (1994) motivate this step.

If we would use an expression like

$$l_i [U_i \cdot x \cdot L_i] r_i$$  \[14\]

to map a particular $U_i \in \{1, n\}$ to $L_i$ when occurring between a left and a right context, $l_i$ and $r_i$, then every $l_i$ and $r_i$ would map substring adjacent to $U_i$.

However, this approach is impossible for the following reason (Kaplan and Kay, 1994): In an example like

$$\{ a \rightarrow b \mid l \mid x \mid . x \} ;$$  \[15\]

where we expect $xaxax$ to be replaced by $xbbbx$, the middle $x$ serves as a context for both $a$’s. A relation described by [14] could not accomplish this. The middle $x$ would be mapped either by an $r_i$ or by an $l_i$ but not by both at the same time. That is why only one $a$ could be replaced and we would get two alternative lower strings, $xaxax$ and $xaaxa$.

Therefore, we have to use the contexts, $l_i$ and $r_i$, without mapping them. For this purpose we introduce auxiliary brackets $<i$ after every left context $l_i$ and $>i$ before every right context $r_i$. The replacement maps those brackets without looking at the actual contexts.

We need separate brackets for empty and non-empty UPPER. If we used the same bracket for both this would mean an overlap of the substrings to replace in an example like $x_{>i}^1 <a_i^1 >a_i^1$. Here we might have to replace $>i <a_i$ and $<a_i >a_i$ where $<a_i$ is part of both substrings. Because of this overlap, we could not replace both substrings in parallel, i.e. at the same time. To make the two replacements sequentially is also impossible in either order, for reasons in detail explained in (Kempe and Kärntunen, 1995).

A regular relation describing replacement in context (and a transducer that represents it), is defined by the composition of a set of “simpler” auxiliary relations. Context brackets occur only in intermediate relations and are not present in the final result.

#### 2.3.2 Preparatory Steps

Before the replacement we make the following three transformations:

1. Complex regular expressions like $[4]$ are transformed into elementary ones like $[5]$, where every single replacement consists of only one UPPER, one LOWER, one LEFT and one RIGHT expression. E.g.

   $$\{ [.] . \rightarrow b \mid l \mid x \mid . y \} ;$$

   $$\{ [.] \rightarrow c \mid \ldots \mid v \mid . w \} ;$$

   would be expanded to

   $$\{ [.] \rightarrow b \mid l \mid x \mid . y \} ;$$

   $$\{ [.] \rightarrow c \mid \ldots \mid v \mid . w \} ;$$

2. Since we have to use different types of brackets for the replacement of empty and non-empty UPPER (cf. 2.3.1), we split the set of parallel replacements into two groups, one containing only replacements with empty UPPER and the other one only with non-empty UPPER. If an UPPER contains the empty string but is not identical with it, the replacement will be added to both groups but with a different UPPER. E.g. $[17]$ would be split into

   $$\{ a \rightarrow b \mid l \mid x \mid . y \} ;$$

   $$\{ e \rightarrow f \mid l \mid v \mid . w \} ;$$

   the group of non-empty UPPER and

   $$\{ [.] \rightarrow b \mid l \mid x \mid . y \} ;$$

   $$\{ [.] \rightarrow c \mid \ldots \mid v \mid . w \} ;$$

   the group of empty UPPER.

3. All empty UPPER of type $[.]$ are transformed into type $[.]$ and the corresponding LOWER are replaced by their Kleene star function. E.g. $[18]$ would be transformed into

   $$\{ [.] \rightarrow b \mid l \mid x \mid . y \} ;$$

   $$\{ [.] \rightarrow c \mid \ldots \mid v \mid . w \} ;$$

   The following algorithm of conditional parallel replacement will consider all empty UPPER as being of type $[.]$, i.e. as not being adjacent to another empty string.

#### 2.3.3 The Replacement itself

Apart from the previously explained symbols, we will make use of the following symbols in the next regular expressions:

$$<_{allE} \{ <_{altE} \ldots <_{altBE} \}, \text{union of all left brackets for empty UPPER.}$$

$$>_{altE} \{ >_{altE} \ldots >_{altBE} \}, \text{union of all right brackets for empty UPPER.}$$

$$<_{allBE} \{ <_{altBE} \ldots <_{altBE} \}, \text{union of all left brackets for non-empty UPPER.}$$

$$>_{altBE} \{ >_{altBE} \ldots >_{altBE} \}, \text{union of all right brackets for non-empty UPPER.}$$

$$<_{alt} \{ <_{alt} \ldots <_{alt} \}, \text{ignore inside operator.}$$

**Example:** $abcxf = [abc/xf] - [x ?*] + [?* x]$, inside the string $abc$, i.e. between $a$ and $b$ and between $b$ and $c$, all $x$ will be ignored any number of times.
We compose the conditional parallel replacement of the six auxiliary relations described by Kaplan and Kay (1994) and Karttunen (1995) which are:

1. InsertBrackets
2. ConstrainBrackets
3. LeftContext
4. RightContext
5. Replace
6. RemoveBrackets

The composition of these relations in the above order, defines the upward-oriented replacement. The resulting transducer maps UPPER inside an input string to LOWER, when UPPER is between LEFT and RIGHT in the input context, leaving everything else unchanged. Other variants of the replacement operator will be defined later.

For every single replacement \{ \( U_i \rightarrow l_i \mid l_i \) \} we introduce a separate pair of brackets \( <_i \) and \( >_i \) with \( i \in \{1E...mE\} \) if UPPER is identical with the empty string and \( i \in \{1...n\} \) if UPPER does not contain the empty string. A left bracket \( <_i \) indicates the end of a complete left context. A right bracket \( >_i \) marks the beginning of a complete right context.

We define the component relations in the following way. Note that UPPER, LOWER, LEFT and RIGHT \( (U_i, L_i, l_i \) and \( r_i \) \) stand for regular expressions of any complexity but restricted to denote regular languages. Consequently, they are represented by networks that contain no fis pairs.

1. InsertBrackets

\[ \begin{align*}
\text{(1)} & \quad l_i \rightarrow <a_i
\end{align*} \]

The relation inserts instances of all brackets on the lower side (everywhere and in any number and order).

2. ConstrainBrackets

\[ \begin{align*}
\text{(2)} & \quad \exists[ >a_i]^\star [ >a_iN^K ] \quad [23] \\
\text{(3)} & \quad \exists[ <a_i]^\star [ <a_iN^K ] \quad [24] \\
\text{(4)} & \quad \exists[ <a_iN^K]^\star [ <a_i ] \quad [25]
\end{align*} \]

The language does not apply to single brackets but to their types and allows them to be only in the following order:

\[ >a_iN^K \rightarrow <a_iN^K ] \quad [26] \]

The composition of the steps (1) and (2) invokes this constraint, which is necessary for the following reasons:

If we allowed sequences like \( <_3 U_3 <_1 U_1 >_1 \) we would have an overlap of the two substrings \( <_3 U_3 >_3 \) and \( <_1 U_1 >_1 \) which have to be replaced. Here, either \( U_1 \) or \( U_3 \) could be replaced but not both at the same time.

If we permitted sequences like \( >_1 E <_2 <_1 B U_2 >_2 \) we would also have an overlap of the two replacements which means we could either replace \( <_2 U_2 >_2 \) or \( >_1 E <_1 B \) but not both.

3. LeftContext

\[ \begin{align*}
\text{(5)} & \quad \lambda_1 \& \ldots \& \lambda_n \quad [27] \\
\text{(6)} & \quad \lambda_i \quad [28]
\end{align*} \]

The composition forces every instance of a left bracket \( <_i \) to be immediately preceded by the corresponding left context \( l_i \) and every instance of \( l_i \) to be immediately followed by \( <_i \), ignoring all brackets that are different from \( <_i \) in between, and all brackets (\( <_i \) included) inside \( l_i \) (\( \lambda_i \)). We separately make the constraints \( \lambda_i \) for every \( <_i \) and \( l_i \) and then intersect them in order to get the constraint for all left brackets and contexts.

4. RightContext

\[ \begin{align*}
\text{(7)} & \quad \forall i \in \{1E...mE\}, \lambda_i = \exists[ l_i >a_i ]^\star [ <a_i ]^\star \quad [29] \\
\text{(8)} & \quad \exists[ l_i >a_i ]^\star [ <a_i ]^\star \quad [30]
\end{align*} \]

The constraint forces every instance of a left bracket \( <_i \) to be immediately preceded by the corresponding right context \( r_i \) and every instance of \( r_i \) to be immediately followed by \( >_i \), ignoring all brackets that are different from \( >_i \) in between, and all brackets (\( >_i \) included) inside \( r_i \) (\( \lambda_i \)). We separately make the constraints \( \lambda_i \) for every \( >_i \) and \( r_i \) and then intersect them in order to get the constraint for all right brackets and contexts.

5. Replace

\[ \begin{align*}
\text{(9)} & \quad R \text{ is a string that does not contain any bracketed \( UPPER \) .} \\
\text{(10)} & \quad N = N_E \& \ldots \& N_m \& N_1 \& \ldots \& N_n \quad [31]
\end{align*} \]

A particular bracketed empty \( UPPER \) \( >_i \) is excluded from the corresponding \( \lambda_i (i \in [1E, mE]) \) by

\[ \begin{align*}
\text{(11)} & \quad N_i = \exists[ l_i >a_i ]^\star [ <a_i ]^\star \quad [32] \\
\text{(12)} & \quad \exists[ l_i >a_i ]^\star [ <a_i ]^\star \quad [33]
\end{align*} \]

The replacement \( R \) of non-empty \( UPPER \) \( U_i \) \( U_i (i \in \{1, n\}) \) is performed by:

\[ \begin{align*}
\text{(13)} & \quad U_i \rightarrow \lambda_i \quad [34] \\
\text{(14)} & \quad \lambda_i \quad [35]
\end{align*} \]

To illustrate this: Suppose we have a set of replacements containing among others:

\[ \begin{align*}
\text{(15)} & \quad a \rightarrow b || x \rightarrow y \\
\text{(16)} & \quad \lambda_i \quad [36]
\end{align*} \]

This particular replacement is done by mapping inside an input string every substring that looks like (underlined part) using the brackets \( <_i \) and \( >_i \) to a substring (underlined part)
\[ ... x >_1 >_E >_E <_E <_1 b >_1 >_2 >_E >_E <_E <_2 y ... \]

The replacement \( R_i \) of empty upper \( U_i \) (\( i \in [1, mE] \)) is performed by:

\[
\begin{align*}
\{ & 0, x, [[>_{\text{aiR}} \quad <_{\text{aiL}} \mid \text{aiNE}] \} \} \# 37 \\
\{ >_{\text{j}} \mid <_{\text{j}} \} \quad \{ & 0, x, \{ L_{\text{j}} >_{\text{aiR}} \} \} \quad \{ <_{\text{j}} \times >_{\text{j}} \} \\
\{ & 0, x, [[>_{\text{aiR}} \quad >_{\text{j}} \mid \text{aiNE}] \} \} \# 38
\end{align*}
\]

In the following example we replace the empty \( U_{2E} \) by \( L_{2E} \). Suppose we have in total one replacement of non-empty upper and two of empty upper, one of which is

\[ \{ . . \} \rightarrow b \mid x-y \; \] 39

This replacement is done by mapping inside a string every substring that looks like (underlined part)

\[ ... x >_1 >_E >_E >_E >_E <_E <_1 y ... \]

using the brackets \( >_2 E <_2 E \) into a substring (underlined part)

\[ ... x >_1 >_E >_E >_E >_E <_E <_1 y ... \]

The occurrence of exactly one bracket pair \( >_E \) and \( <_E \) between a left and a right context, actually corresponds to the definition of a (single) empty string expressed by [. . ] (cf. sec. 2.2).

The brackets \( >_{>_{E}} >_{<_{E}} <_{<_{E}} <_{<_{E}} \) and \( >_{>_{E}} >_{<_{E}} <_{<_{E}} <_{<_{E}} \) in [40] are inserted on the lower side any number of times (including zero), i.e. they exist optionally, which makes them present if checking for the left or right context requires them, and absent if they are not allowed in this place.

This set of brackets does not contain those ones used for the replacement, \( >_{<_{E}} \), because if we later check for them we do not want this check to be always satisfied but only when the specified contexts are present, in order to be able to confirm or to cancel the replacement a posteriori.

This set of optionally inserted brackets equally does not contain those which potentially could be used for the replacement of adjacent non-empty strings, i.e. \( >_{\text{aiR}} \) on the left and \( <_{\text{aiNE}} \) on the right side of the expression. Otherwise, checking later for the legitimacy of the adjacent replacements would no longer be possible.

(6) RemoveBrackets

\[ \{ <_{\text{aiL}} \rightarrow [ \} \] 41

The relation eliminates from the lower-side language all brackets that appear on the upper side.

3 Variants of Replacement

3.1 Application of context constraints

We distinguish four ways how context can constrain the replacement. The difference between them is where the left and the right contexts are expected, on the upper or on the lower side of the relation, i.e. LEFT and RIGHT contexts can be checked before or after the replacement.

We obtain these four different applications of context constraints (denoted by \( [ \}, \{ \), \( \{ \), \( \{ \) and \( \langle \), \( \langle \) and \( \rangle \) by varying the order of the auxiliary relations (steps (3) to (5)) described in section 2.3.3 (cf. [22]):

(a) Upward-oriented

\[
\begin{align*}
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots \\
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots 
\end{align*}
\]

(b) Right-oriented

\[
\begin{align*}
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots \\
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots 
\end{align*}
\]

(c) Left-oriented

\[
\begin{align*}
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots \\
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots 
\end{align*}
\]

(d) Downward-oriented

\[
\begin{align*}
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots \\
\{ & U_i \rightarrow L_i \mid l_i - r_i \} \mid \ldots 
\end{align*}
\]

The versions (a) to (c) roughly correspond to the three alternative interpretations of phonological rewrite rules discussed in Kaplan and Kay (1994). The upward-oriented version corresponds to the simultaneous rule application; the right- and left-oriented versions can model rightward or leftward iterating processes, such as vowel harmony and assimilation.

In the downward-oriented replacement the operation is constrained by the lower (left and right) context. Here the \( U_i \) get mapped to the corresponding \( L_i \) just in case they end up between \( l_i \) and \( r_i \) in the output string.

3.2 Inverse, bidirectional and optional replacement

Replacement as described above, \( \rightarrow \), maps every \( U_i \) on the upper side unambiguously to the corresponding \( L_i \) on the lower side but not vice versa. A \( L_i \) on the lower side gets mapped to \( U_i \) or \( U_i \) on the upper side.

The inverse replacement, \( \leftarrow \), maps unambiguously from the lower to the upper side only. The bidirectional replacement, \( \leftrightarrow \), is unambiguous in both directions.

Replacements of all of these three types (directions) can be optional, \( \rightarrow \) \( \leftarrow \) \( \leftrightarrow \), i.e. they are either made or not. We define such a relation by changing \( N \) (the part not containing any bracketed upper) in expression [28] into \( ? \) that accepts every substring:

\[ [ ?* R_e ] ?* \] 46

Here an \( U_i \) is either mapped by the corresponding \( R_i \) contained in \( R \) (cf. [32]) and therefore replaced by \( L_i \) or it is mapped by \( ? \) and not replaced.
a stem in -iss- (e.g.: finir/finissent) rather than from
a more general root (e.g.: finir) because once this
stem is assumed, the subjunctive ending itself be-
comes completely regular:

(that I finish) (that I run)
que je finisse-que je cour-
quie tu finisses-que tu cour-
.....
quie ils finissent-que ils cour-

The algorithm we propose here, is straightfor-
ward: We first derive the present subjunctive stem
from the third person plural present indicative
(e.g.: finiss, cour), then append the suffix corre-
sponding to the given person and number.

The first step can be described as follows:

\[
\text{define LETTER : } a \cup b \cup c \cup d \cup \ldots;
\]
\[
\text{define SUFFIX : SubjP..[SG]..[P3]..[Verb]..;}
\]
\[
\text{define StemRegular : [1 .. ] <-> IndP PL P3 Verb | | LETTER | TAG}
\]
\[
\text{LexInd TAG+}
\]
\[
[..] <-> IndP PL P3 Verb
\]
\[
\text{e n t } <-> SUFFIX | | - TAG ;}
\]

The first transducer in [49] inserts the tags of the
third person plural present indicative between the
word and the tags of the actually required subjunc-
tive form. The second transducer in [49] which is an
indicative lexicon of -ir verbs, concatenated with a
sequence of at least one tag, provides the indica-
tive form and keeps the initial subjunctive tags.
The last transducer in [49] replaces the suffix -ent
by the symbol SUFFIX. E.g.:

\[
\text{finir} \quad \text{finiss} \quad \text{finissent} \quad \text{finissent}
\]

To append the appropriate suffix to the subjunc-
tive stem, we use the following transducer which
maps the symbol SUFFIX to a suffix and deletes all
tags:

\[
define SUFFIX : [ \{ SUFFIX \to e | - TAG* [SG [P1 [P3]] },
{ SUFFIX \to e as | - TAG* [SG [P2 ]},
{ SUFFIX \to io ns | - TAG* PL P1 },
{ SUFFIX \to io z | - TAG* PL P2 },
{ SUFFIX \to e n t | - TAG* PL P3 } ]
\]
\[
\text{TAG } \to [ ]
\]

The complete generation of subjunctive forms can
be described by the composition:

\[
define LexSubjP : StemRegular . SUFFIX ;
\]

The resulting (single) transducer LexSubjP re-
preents a lexicon of present subjunctive forms of
French verbs ending in -ir. It maps the infinitive of
those verbs followed by a sequence of subjunctive
tags, to the corresponding inflected surface form
and vice versa.

All intermediate transducers mentioned in this
section will contribute to this final transducer but
will themselves disappear.

The regular expressions in this section could also
be written in the two-level formalism (Kosken-
niemi, 1983). However, some of them can be ex-
pressed more conveniently in the above way, espe-
cially when the replace operator is used.

E.g., the first line of [49], written above as:

\[
(1 \to ) \leftrightarrow IndP PL P3 Verb | | LETTER | TAG
\]

would have to be expressed in the two-level formal-
ism by four rules:

\[
0 : IndP <-> LETTER | ( [PL] [P3] ) [Verb] | TAG
0 : PL <-> LETTER | ( [IndP] ) [PL] [Verb] | TAG
0 : P3 <-> LETTER | ( [IndP] ) [PL] - [Verb] | TAG
0 : Verb <-> LETTER | ( [IndP] ) [PL] [P3] - TAG
\]

Here, the difficulty comes not only from the large
number of rules we would have to write in the above
example, but also from the fact that writing one of
these rules requires to have in mind all the others,
avoid inconsistencies between them.

Acknowledgements

This work builds on the research by Ronald Kaplan
and Martin Kay on the finite-state calculus and the
implementation of phonological rewrite rules (1994).

Many thanks to our colleagues at PARC and RXRC
Grenoble who helped us in whatever respect, partic-
ularly to Annie Zaenen, Jean-Pierre Chauod, Marc
Dymetman, Kenneth Beeley and Anne Schiller for
helpful discussion on different topics, and to Irene
Maxwell for correcting the paper.

References

Brill, Eric (1992). A Simple Rule-Based Part of Speech
Tagger. Proc. 3rd conference on Applied Natural
Language Processing. Trento, Italy, pp. 152-155.

Kaplan, Ronald M., and Kay, Martin (1981). Phono-
logical Rules and Finite-State Transducers. Annual
Meeting of the Linguistic Society of America. New
York.

Kaplan, Ronald M. and Kay, Martin (1994). Regular
Models of Phonological Rule Systems. Computational
Linguistics. 20:3, pp. 331-378.

Karlsson, Fred, Voutilainen, Atro, Heikkilä, Juhu,
and Anttila, Arto (1994). Constraint Grammar:
A Language-Independent System for Parsing Unre-
stricted Text. Mouton de Gryuter, Berlin.

Karttunen, Lauri (1995). The Replace Operator. Proc.
ACL-95. Cambridge, MA, USA. cmp-lg/9504032

Kempe, Andre and Karttunen, Lauri (1995). The Par-
allel Replacement Operation in Finite State Calculus.
Technical Report MLTT-021. Rank Xerox Research
Centre, Grenoble Laboratory. Dec 21, 1995.
http://www.xerox.fr/grenoble/mltt/reports/home.html

Koskenniemi, Kimmo (1983). Two-Level Morphol-
ogy: A General Computational Model for Word-Form
Recognition and Production. Dept. of General Lin-
guistics. University of Helsinki.

Koskenniemi, Kimmo (1990). Finite-State Parsing and
Disambiguation. Proc. Coling-90. Helsinki, Finland.

Koskenniemi, Kimmo, Tapanainen, Pasi, and Voutil-
ainen, Atro (1992). Compiling and using finite-state
syntactic rules. Proc. Coling-92. Nantes, France.

Roche, Emmanuel and Schabes, Yves (1995). De-
terministic Part-of-Speech Tagging with Finite-State
Transducers. Computational Linguistics. 21, 2, pp.
227-53.

Voutilainen, Atro (1994). Three Studies of Grammar-
Based Surface Parsing of Unrestricted English Text.
The University of Helsinki.