*Natural Language Engineering* **9** (1): 65–85. © 2003 Cambridge University Press
DOI: 10.1017/S1351324903003103 Printed in the United Kingdom


# *Logical specification of regular relations for NLP*

NATHAN VAILLETTE

*Department of Linguistics, Ohio State University, 222 Oxley Hall, 1712 Neil Avenue, Columbus, OH, 43210-1298, USA*
*e-mail*: nvail@ling.ohio-state.edu

(*Received 2 April 2002; revised 8 November 2002*)

**Abstract**

This paper describes how the use of monadic second-order logic for specifying regular languages can be extended for specifying regular relations, providing a declarative description language for finite state transductions of the sort used in NLP. We discuss issues arising in the integration into an automaton toolkit of an implementation of the conversion from logic formulas to automata. The utility of the logic of regular relations is demonstrated by showing how it can be used to define the family of replacement operators in a way that lends itself to straightforward proofs of correctness.

## 1 Introduction

Regular languages and relations can be encoded using regular expressions, which can be compiled to finite state machines.[1] This frees us from the need to work with the machines directly, thus providing a higher-level description language. However, this kind of description is not always the most convenient. It would also be useful to be able to describe regular sets in a logic. In many situations, this would be closer to the thought process and would provide a kind of executable specification for automata. Such a declarative approach is also advantageous for verification of correctness.

Monadic Second-Order Logic (MSOL) provides an expressive framework with an important connection to regular sets. Theoretical work on the decidability of restrictive theories of arithmetic in Büchi (1960) and Elgot (1961) established the equivalence of MSOL with regular sets, and thus with finite state automata. Applications of the theory were not undertaken for some time, because of its extremely daunting complexity: in the worst case, the space requirements of the conversion from logic formulas to the corresponding finite state automata is bounded from below by a stack of exponentials proportional in size to the number of quantifiers in the formula (Meyer 1975). However, work on recent implementations such as the

[1] See Hopcroft and Ullman (1979) for definitions of regular languages and finite state automata. Good overviews of regular relations from an NLP perspective can be found in Kaplan and Kay (1994) and the introduction of Roche and Schabes (1997).

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

the MONA (Klarlund and Møller 2001) and MOSEL (Kelb, Margaria, Mendler and Gsottberger 1997) projects have demonstrated that the worst case need not be a problem in practice. In this paper, we will describe the use of MSOL for characterizing regular relations (on strings) for NLP purposes. Section 6 shows this logic in action in characterizing finite state replacement transductions.

## 2 Notation

A *string* $s$ over a finite alphabet $\Sigma$ is an element of the free monoid $\Sigma^*$ generated by $\Sigma$. The unit of this monoid is the empty string, written $\epsilon$, and the operation is written as $a \cdot b$, or simply as the concatenation $ab$ when no confusion can arise. A *substring* of $s$ is any $s'$ such that $s = vs'w$ for some strings $v$ and $w$. The length of $s$ is written $|s|$.

A string of length $n$ can also be viewed as a function from $n$ to $\Sigma$, where $n$ is equated with the set of all natural numbers less than $n$. For $i \leq j$, we use the half-open interval notation $[i, j)$ for $\{k \in N \mid i \leq k < j\}$. Note that $[i, i) = \emptyset$. The notation $s[i, j)$ denotes the substring $s(i) \cdots s(j-1)$ if $j > i$ and denotes $\epsilon$ if $i = j$.

For a string $s$ of length $n$, we speak of a *position* in $s$ meaning any $k < n$ and say that symbol $s(k)$ *occupies* position $k$ in $s$. Similarly, we speak of a *region* of $s$ meaning any interval in the domain of $s$. It is sometimes convenient to extend a string of length $s$ so that $s(n)$ is also defined. We assume $s(n) = \#$, where we presuppose $\# \notin \Sigma$. We ambiguously use $\#$ to also refer to the position $n$ itself. This extension allows us to refer to a suffix of $s$ with the open interval notation as a substring $s[i, \#)$ $(= s[i, n))$.

We presuppose familiarity with the basic concepts of regular languages, regular relations, and finite state automata and transducers (see footnote 1 for references). We use the notation in Table 1 for regular expressions, given in order of operator precedence.[2]

One particular concept that will be of use to us is that of a same length relation. A regular relation $R \subseteq \Sigma^* \times \Sigma^*$ is a *same length relation* if for every $\langle w, w' \rangle \in R$, $|w| = |w'|$. A same length relation can also be viewed as a regular language over the alphabet $\Sigma \times \Sigma$ of pairs of symbols. An element $p = \langle w, w' \rangle$ of $R$ can thus be viewed as a string $p(0) \ldots p(n-1)$ where $p(i) = \langle w(i), w'(i) \rangle$ for all $i < n$.

## 3 Logical background

### 3.1 Logic and automata

The exploration of the connection between logic and regular languages began with Büchi (1960). Specifically, Büchi was concerned with the theory of weak second-order

[2] Note that relation operators also apply to languages, construed as relations of arity one. The same length cross product $A \times\!\!\times B$ denotes $\{\langle s, s' \rangle \in A \times B \mid |s| = |s'|\}$. The composition $A \circ B$ denotes $\{\langle s, s' \rangle \mid$ for some $s''$, $sAs''$ and $s''Bs'\}$ (note that many authors write the arguments of $\circ$ in the reverse order).

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

Table 1. *Regular expression operators*

| | |
|---|---|
| empty string: | $\epsilon$ |
| empty language: | $\emptyset$ |
| complement of language $A$: | $\neg A$ |
| Kleene closure of relation $A$: | $A^*$ |
| concatenation of relation $A$ and $A^*$: | $A^+$ |
| concatenation of relations $A$ and $B$: | $AB$ or $A \cdot B$ |
| $A \cdot A \cdots A$ $k$ times: | $A^k$ |
| identity relation on language $A$: | id($A$) |
| domain of relation $A$: | dom($A$) |
| range of relation $A$: | ran($A$) |
| set difference of languages $A$ and $B$: | $A - B$ |
| intersection of languages $A$ and $B$: | $A \cap B$ |
| union of relations $A$ and $B$: | $A \cup B$ |
| cross product of languages $A$ and $B$: | $A \times B$ |
| same length cross product of languages $A$ and $B$: | $A \lll$ B |
| $A \times A \times \cdots \times A$ $k$ times: | $A^{(k)}$ |
| composition of relations $A$ and $B$: | $A \circ B$ |

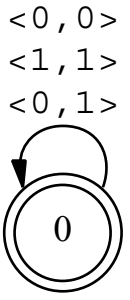



Fig. 1. Automaton for $X \subseteq Y$.

arithmetic (W.2.A). This theory is expressed in a language which has monadic second order variables that are interpreted as finite sets of natural numbers.

Concentrating on formulas with only free predicate variables, an interpretation of a formula is an assignment of finite sets of natural numbers to the free variables of the formula. Since the sets are finite, for any interpretation there is some maximal $n$ s.t. every $m > n$ is in none of the sets. So an interpretation of a theory of W.2.A amounts to a specification for each number $i$ from 0 to $n$ of whether $i$ is in the denotation of free variable 1, whether $i$ is in the denotation of free variable 2, and so forth for all the $k$ free variables. Thus, an interpretation corresponds transparently to a finite string of $k$-tuples of truth values 0 and 1, where the $j$-th projection of tuple $i$ is 1 iff $i$ is in the denotation of the $j$-th free variable. What Büchi proves is that for any formula of W.2.A, there is an automaton over $k$-tuples of truth values that accepts only satisfying interpretations of the formula, and vice versa.

As an example, the automaton for the two variable formula $X \subseteq Y$ is given in in figure 1. Here 0 is the initial state, and final states are represented by double circles. The first element of a tuple represents the $X$ tape and the second element the $Y$ tape.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

Applications of Büchi's result did not appear until the 1990s. A number of tools implementing the conversion from logic to automata now exist, some not only for W.2.A but also for extensions to it that provide a logic for finite trees. These include MONA (Klarlund and Møller 2001), MOSEL (Kelb, Margaria, Mendler and Gsottberger 1997), SHASTA, and Glenn and Gasarch (1997). The primary reason for the 30 year delay between theory and concrete application seems to have been the discouraging complexity of the conversion mentioned in the introduction. This complexity, however, need not be a problem. All the implementations above but the last use special techniques involving Binary Decision Diagrams (BDDs) for representing the automata that reduce their size (e.g. see Klarlund and Møller (2001) for discussion). Of course, this technique doesn't affect the worst case. Nonetheless, it seems that the kinds of formulas used in practice generally don't cause a space blow-up. Even without BDDs, the implementation described in section 6 takes only seconds to compile the fairly complex formulas used in the replacement example of section 6 using over a dozen quantifiers and dozens of connectives. The reader is referred especially to Klarlund and Møller (2001) for more arguments for the feasibility of MSOL in practice.

### *3.2 MSO(Str)*

For our purposes, we want to use MSOL for characterizing regular languages (and in section 4, relations) over any finite alphabet of symbols. We would like to use logical statements that say, for example, that certain symbols occupy certain string positions. The language MSO(Str), monadic second-order logic for strings (e.g. see Thomas (1996)), provides such expressivity, and gives us second-order quantification over sets of positions in a string. The syntax is as follows. We take as given denumerable sets of first-order variables $\{x, y, z, \ldots\}$ and second-order variables $\{X, Y, Z, \ldots\}$. For any first-order variables $v$ and $v'$, for any second-order variables $V$ and $V'$, for any $\sigma \in \Sigma$, we can form the atomic formulas $symbol(v, \sigma)$, $V \subseteq V'$, $v \in V$, and $succ(v, v')$. The molecular formulas are (for any formulas $\phi$ and $\psi$, for any variables $v$ and $V$): $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\exists v(\phi)$,$\exists V(\phi)$, $\forall v(\phi)$, and $\forall V(\phi)$.

The intended interpretation can be described informally as follows. First-order variables range over positions in a string, and second-order variables over sets of such positions. A closed formula states a restriction on positions and their occupants and denotes the set of all strings obeying the restriction. $symbol(v, \sigma)$ says that the symbol $\sigma$ occupies the position $v$. $v \in V$ states a set membership. Finally, $succ(v, v')$ says that position $v$ directly precedes position $v'$. Connectives and quantifiers have their usual classical meaning.[3]

It is convenient to introduce some constant and function symbols as shorthand, such as: $\emptyset$ (the empty set), set operations $\cup$ and $\cap$, $<$ and $\leq$ (linear precedence on positions), 0 (the first position), $\min(V)$ and $\max(V)$ (with $\min(\emptyset) = \max(\emptyset) = 0$), $string(V)$ which claims that $V$ is a continuous string, and \$ (the set of all positions).

[3] A presentation of the model theory for MSO(Str) can be found in Ebbinghaus and Flum (1999, Ch. 6).

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

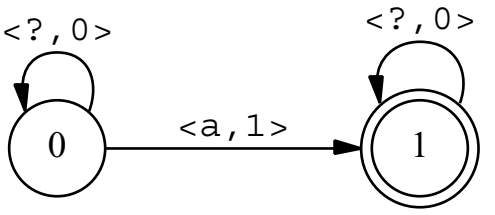



Fig. 2. Automaton for $symbol(x, a)$.

As an example of a definable predicate, *succ2* generalizes *succ* to substrings. $succ2(V, V')$ holds iff $V$ directly precedes $V'$ with no intervening positions. It can be defined as in (1):

(1) $succ2(V, V') \equiv_{\text{def}}$
$string(V) \wedge string(V') \wedge \forall x, y(x \in V \wedge y \in V' \rightarrow x < y) \wedge$
$\neg\exists x, y, z(x \in V \wedge y \in V' \wedge z \notin V \wedge z \notin V' \wedge x < z < y)$

The translation of MSO(Str) formulas to automata works similarly as for W.2.A. A formula with $k$ free variables corresponds to an automaton over $k+1$-tuples, with an extra tape added to carry information about the alphabetic symbols. For instance, the automaton for $symbol(x, a)$ shown in figure 2 requires exactly one position on the $x$ tape (the second element of the tuple) to contain a 1, while that same position on the symbol tape (the first element of the tuple) contains an $a$. Every other position has a 0 on the $x$ tape and can have any symbol (represented by "?") on the symbol tape. A closed formula corresponds to a one tape automaton over symbols, which recognizes a regular language over $\Sigma$.

### *3.3 Logic and regular expressions*

Any regular language can be described by a closed formula of MSO(Str). Nonetheless, it is convenient to extend the syntax of the logic to include the syntax of (extended) regular expressions and add a new predicate symbol *matches* that takes as arguments a variable and a regular expression. For the regular expression $\mathscr{E}$, $matches(X, \mathscr{E})$ will hold if the interpretation of $X$ is a substring which is an element of the language denoted by $\mathscr{E}$.

The automaton for $matches(X, \mathscr{E})$ requires that there be a substring for which the symbol-tape component is in $\mathscr{E}$ and the $X$-tape component contains all 1s. This automaton can be built directly from the automaton for $\mathscr{E}$. For instance, the automaton for $matches(X, a^+b)$ is given in figure 3.

As an example of the intuitive advantage of logical description, notice how *matches* and *succ2* allow for a simple characterization of context restrictions.

(2) $\forall X(matches(X, ab) \rightarrow \exists Y(matches(Y, c^*d) \wedge succ2(X, Y)))$

This example states an occurrence restriction, that every occurrence of the language $ab$ must be directly followed by an occurrence of $c^*d$. This can also be expressed (and more compactly) by the extended regular expression $\neg(\Sigma^* \cdot ab \cdot \neg(c^*d \cdot \Sigma^*))$.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

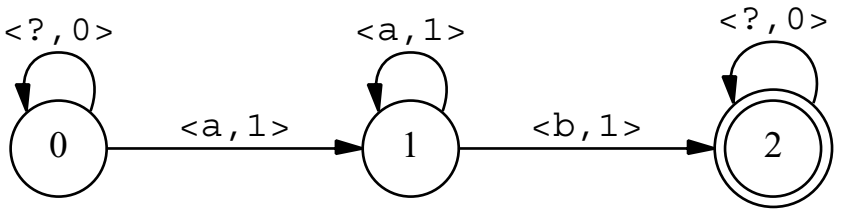



Fig. 3. Automaton for $matches(X, a^+b)$.

However, most people find the logic formula more transparent. In fact, when using such regular expressions to define operators stating context restrictions in their paper on finite state replacement transductions, Kaplan and Kay elucidate them by noting that "the double complementation in the definitions of these ... operators ... constitutes an idiom for expressing universal quantification" (Kaplan and Kay 1994 p. 345). This highlights the naturalness of logical concepts for such expressions and shows the advantage of a specification language in which such concepts can be used directly instead of having to be derived.

We mention here that in certain contexts it is convenient to have a tertiary version of *matches* which takes instead of a second order variable, two first order ones giving the endpoints of a half-open interval: $s$ satisfies $matches(x, y, \mathscr{E})$ w.r.t. a variable assignment $g$ iff $s[g(x), g(y))$ is in $\mathscr{E}$. We use this version in section 6.

## 4 A logic for (some) regular relations

We would like to be able to extend our logic to also describe regular relations. (In this paper, we concentrate on the case of binary relations, but the methods discussed extend to arbitrary arities.) Unfortunately, there is a fundamental problem. Our logic must have conjunction and negation, and these correspond to intersection and complementation of the corresponding languages/relations. However, the class of regular relations is known not to be closed under these operations (see the introduction to Roche and Schabes (1997)).

We pursue the solution of restricting the logic to describe a proper subset of regular relations which is closed under these operations: the same length relations. Since same length relations can be viewed as regular languages over the free monoid $(\Sigma \times \Sigma)^*$, their boolean closure follows immediately.

For same length relations, we can think of a pair of strings as a string of pairs and use the logic to constrain the appearance of these pairs. What we can do is write formulas like $\exists X(matches(X, (abc \times efg)^+))$. This describes the relation where for every input string there is a $k > 0$ such that it contains a region matching $(abc)^k$ while its corresponding output strings contain $(efg)^k$ in the same region. The logic of same length relations used in the rest of this paper is thus nothing more than the application of MSO(Str) to the case where the strings described happen to be elements of $(\Sigma \times \Sigma)^*$.

This notion has strong affinities with two-level morphology (Koskenniemi 1983), which works by constraining the relationship between underlying and surface strings

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

on a symbol-by-symbol basis. What we cannot say in a logic of same length relations is e.g. $\exists X(matches(X,(abc \times ef)^+))$, since $(abc \times ef)^+$ is not a same length relation.

### *4.1 0s and alignments*

The restriction to same length relations seems rather severe and may appear to undermine the usefulness of the logic. However, this need not be the case. In section 6, we describe an application and illustrate the method of first creating a version of it for same length relations and then making simple modifications which generalize it to the full class of regular relations. The generalization relies on another idea from two-level morphology: using an explicit symbol `0` to stand for the empty string, which appears 'internally' but can be so to speak 'hidden from the user'.

Although there are non same length relations which are not directly representable in the logic, it is still possible to characterize such a relation indirectly by using the special symbol $0 \notin \Sigma$ to 'pick up the slack.' For instance, $ab \times c$ can be represented as $ab \times c0$, or alternatively as $ab \times 0c$. The 0 is an actual symbol of the extended alphabet $\Sigma \cup \{0\}$ (which we will write as $\Sigma_0$), but one which can stand proxy for the empty string.

More explicitly: let $\gamma$ be the monoid morphism from the free monoid $(\Sigma_0 \times \Sigma_0)^*$ to $\Sigma^* \times \Sigma^*$ that for $\sigma, \tau \in \Sigma$ maps $\langle \sigma, 0\rangle$ to $\langle \sigma, \epsilon\rangle$, $\langle 0, \tau\rangle$ to $\langle \epsilon, \tau\rangle$, $\langle 0, 0\rangle$ to $\langle \epsilon, \epsilon\rangle$, and any other $\langle \sigma, \tau\rangle$ to itself. For instance, $\gamma(ab \times 0c) = ab \times c$. For any $s \in \Sigma^* \times \Sigma^*$, we say an *alignment* of $s$ is an element of $\gamma^{-1}(s)$, the inverse image of $s$ under $\gamma$. alignments($s$) refers to the set of all alignments of $s$, i.e. $\gamma^{-1}(s)$. Similarly, for $R \subseteq \Sigma^* \times \Sigma^*$, alignments($R$) $= \cup_{s \in R}$alignments($s$). An *alignment of a relation* $R$ is a subset $R'$ of alignments($R$) such that for every $s \in R$, there is at least one alignment of $s$ in $R'$. For instance, given a regular relation $R$, any transducer for $R$ can be made to represent an alignment of $R$ by replacing every transition over $\langle \sigma, \epsilon\rangle$ with one over $\langle \sigma, 0\rangle$ and replacing every transition over $\langle \epsilon, \sigma\rangle$ with one over $\langle 0, \sigma\rangle$. The machine can then be seen as an automaton accepting a subset $S$ of $(\Sigma_0 \times \Sigma_0)^*$ such that $S$ is an alignment of $R$. Note that $S$ is itself a regular language (or same length regular relation).

In some cases, we could let the logic refer to the same length relation alignments($R$) in lieu of a non same length regular $R$; $R$ can be recovered using $\gamma$ (which is implementable as an operation on finite state machines). alignments($R$) acts just like $R$ in the sense that alignments($R$) $\cap$ alignments($S$) $=$ alignments($R \cap S$), and $(\Sigma_0 \times \Sigma_0 - \{\langle 0, 0\rangle\}) -$ alignments($R$) $=$ alignments($(\Sigma^* \times \Sigma^*) - R$). However, this usage presupposes that alignments($R$) is itself a regular relation, which it need not be. In fact, we can prove that alignments($R$) is regular iff $R$ is a finite union of crossproducts of regular languages.[4] Not all regular relations are thus decomposable. For instance, the relation $(ab \times c)^*$ is not.

[4] These are just the recognizable relations; e.g. see Berstel (1979) for definitions. The proof of this equivalence relies on the existence of a morphism from a regular $R$ with regular alignments($R$) to the transition monoid of the minimal automaton recognizing alignments($R$). Unfortunately, recognizability is not a decidable property of regular relations.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

Even for such relations, we can still use a *particular* alignment to stand proxy, e.g. $(ab \times c0)^*$ or $(ab \times 0c)^*$ for $(ab \times c)^*$. We cannot however expect these to act exactly the same as their $\gamma$ images. For instance, $\gamma((ab \times c0)^*) = \gamma((ab \times 0c)^*) = (ab \times c)^*$ and $\gamma(ab \times c0) \in (ab \times c)^*$; however, $(ab \times c0) \cap (ab \times c0)^* = (ab \times c0)$ while $(ab \times 0c) \cap (ab \times c0)^* = \emptyset$. In section 6.6, we shall see how 0s can be put to use despite this limitation.

## 5 Implementation

A simple implementation[5] of the conversion of MSO(Str) formulas to automata has been carried out within the Finite State Utilities (FSU), an open-source finite state machine toolkit written in Prolog (van Noord, n.d.; van Noord and Gerdemann 1999). The FSU includes facilities for manipulating finite automata and transducers, such as visualization, recognition and transduction algorithms, and the conversion of extended regular expressions to machines. The regular expression operators can be extended with macros in Prolog syntax which can call Prolog code. Automata are represented directly in our implementation, not using binary decision diagrams. We have found it advantageous to integrate the logic into such a tool for extensible regular expressions. Logic formulas can be embedded into regular expressions, and any extended or defined regular expressions can be embedded into logic formulas using *matches*. This is very useful, for example, for the definition of *matches-weak-dom* in section 6.6, which can just be defined as a macro in terms of regular expressions and the *matches* predicate.

For comparison, the replacement application in section 6 was also implemented in the MONA system. MONA is based on W.2.A, not MSO(Str). Since association of alphabet symbols with positions is not a feature of W.2.A, it is necessary to represent symbols with extra second order variables. A two-element alphabet can be represented with just a single variable whose interpretation contains exactly the positions occupied by one of the symbols, all of the other positions in $ bearing the other symbol. Similarly, the 8-bit ASCII alphabet requires eight variables, and same length transducers over this alphabet require 16 (see Klarland and Møller (2001) for discussion).

The disadvantage of MONA for describing regular relations is that it is not integrated into a general regular expression toolkit, unlike our implementation in the FSU. All regular expressions in uses of *matches* must be expanded out into statements about the ASCII bit variables in a preprocessing step. Furthermore, the naïve representation of alphabets is not optimal. For instance, the replacement automata of section 6 (even when trim) require a bifurcation which sends some input along one path and all other symbols along another. The specification of, for example, "all symbols but $q$" in the 8-bit ASCII representation results in transitions for 255 other symbols. Most implementations use a more compact representation,

[5] This is available, along with the MONA-based implementation discussed below, from `http://www.ling.ohio-state.edu/~nvail/msol.html`

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

such as transitions over a '?' symbol (assuming $? \notin \Sigma$) for 'all other symbols,' or transitions over explicit negations in the FSU.

For these reasons, we conclude that, despite the arguments of Klarland (1999) for simulating MSO(Str) in W.2.A, the most practical design for an MSO(Str) implementation involves embedding the logic to automaton conversion into a finite state toolkit with *matches* implemented as a basic predicate (i.e. with the automaton for $matches(X, \mathscr{E})$ constructed directly from the automaton for $\mathscr{E}$) able to match any regular expressions of the toolkit. However, unlike our FSU implementation, it would be desirable in a future implementation to use the BDD technology for efficiency.

## 6 The replace operator

This section demonstrates the use of the logic of regular relations by showing how it can be used to define the family of replacement operators in a way that both is transparent and also lends itself to straightforward proofs of correctness.

The replace operator maps strings of $\Sigma^*$ to strings of $\Sigma^*$. It has the form $\phi \rightarrow \psi / \lambda __ \rho$ for $\phi, \psi, \lambda, \rho \subseteq \Sigma^*$. It replaces occurrences of $\phi$ in the input which occur between $\lambda$ and $\rho$ with $\psi$. (We write $\phi \rightarrow \psi$ when $\lambda = \rho = \epsilon$.) It corresponds to the rewrite rules of early generative phonology (see Chomsky and Halle (1968) (SPE)), and is useful not only in computational phonology and morphology, but anywhere text processing requires conditional string replacements. Johnson (1972) was the first to notice that this operator is indeed a regular relation, as long as it does not reapply to its own output (a restriction generally obeyed by SPE rules).

Several varieties of finite state replacement operators along with algorithms for conversion to transducers have been proposed, such as in Gerdemann and van BNoord (1999), Kaplan and Kay (1994), Karttunen (1996, 1997), Kempe and Karttunen (1996) and Mohri and Sproat (1996). All of these implementations have, to varying degrees, a procedural nature, based on a cascade of transducers where special marker symbols are inserted to delimit regions and are later deleted. They can become rather complicated, and it is difficult to rigorously prove their correctness. However, we can give a declarative definition in the logic which has a clear connection to the intended interpretation and a transparent correctness proof.

For simplicity and generality, we actually discuss the notion of replacement with 'backreferences' presented in Gerdemann and van Noord (1999). Here, instead of specifying the input-output relationship with an input $\phi$ and output $\psi$, we can use an arbitrary regular relation $T$. We now write $T / \lambda __ \rho$, which replaces any $s \in \mathrm{dom}(T)$ which is between $\lambda$ and $\rho$ with $T(s)$. Thus $\phi \rightarrow \psi / \lambda __ \rho$ corresponds to the special case where $T$ is a cross-product $\phi \times \psi$, but transductions not definable as cross-products are now also possible. For example, $(a \times b)^+ / c __ c$ replaces a string of $k$ $a$s between $c$ and $c$ with a string of $k$ $b$s.

Different versions of the replace operator place different restrictions on what can and must be replaced. The 'earliest-first' replacement strategy is the version most easily understood in algorithmic terms. Informally, for a replacement $\phi \rightarrow \psi / \lambda __ \rho$ the algorithm reads the input string from left to right. Where no replacement is

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

possible, it rewrites the input unchanged. It is required to make a replacement at the first point where, having just outputed $\lambda$, it encounters $\phi\rho$ in the following input. At that point, instead of $\phi$, it outputs $\psi$, and then continues to process the string in the same way starting at the beginning of $\rho$, the next replacement being made at the earliest possible subsequent point. Importantly, the context and input can interact. The preceding $\lambda$ may have been present in the input, or may instead have resulted from an earlier replacement. Similarly, part or all of the following $\rho$ may later get replaced and not appear as such in the final output.[6]

### *6.1 Undirected replacement*

The algorithm described informally in the previous section is nondeterministic, since different replacements of different lengths may be possible at different points. For example, in $a^+ \rightarrow b$ when given a string of $k$ $a$s, the input expression $a^+$ may capture any non-empty prefix. The output can be any number of $b$s from 1 (capturing the whole input as one replacement) to $k$ (capturing each $a$ separately.). One way to strengthen the conditions to yield a deterministic version is discussed in section 6.3, requiring that the longest possible replacement be made at each point.

On the other hand, the algorithm can be weakened by removing the constraint that the earliest match be replaced. For instance, $ab \cup bc \rightarrow d$ will map $abc$ only to $dc$ if the earliest match is required, but if not, will also allow $ad$. What is still prohibited is the output $abc$; the reason is that there is a region of the input (the $ab$ or $bc$ substring) which could've been replaced (it was indeed a candidate, and no other replacement disrupted its input or context) but was not.

For our logical definition, we will start with the version without earliest matching, what Karttunen (1996) calls 'undirected' replacement, relying on the informal characterization above. More specific versions of the operator can be seen as deriving from the undirected version by the addition of further constraints. We discuss a specific example of directed replacement in section 6.3. In particular, we give an algorithmic definition of earliest/longest match replacement and prove its equivalence to a logical definition.

In the next section we come to the definition of undirected replacement in the logic. Recall that our logic of regular relations can only describe same length relations. Therefore, our logical definition of replacement will at first be restricted to cases where the input and output are the same length. Section 6.6 then discusses how to ease this restriction.

### *6.2 Undirected replacement: logical definition*

An informal definition of the regular relation $R \subseteq (\Sigma \times \Sigma)^*$ corresponding to a replacement $T/\lambda_\rho$ can be given as follows. For expository reasons, we will first

[6] Because of the conception of the replacement as processing the string from left to right, the left context is understood as part of the output and the right context as part of the input. This is what Karttunen (1996) calls 'rightward orientation.' We stick to this orientation here for concreteness, but other orientations are equally possible; see Karttunen's discussion for details.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

describe the unconditional version, where no left and right context are checked. In this context, we use the term *replacement-candidate* as follows. For a substring $s$ of a word $w \in (\Sigma \times \Sigma)^*$, $s$ is a replacement-candidate if $\mathrm{dom}(s) \in \mathrm{dom}(T)$.

### *6.2.1 Unconditional replacement*

To define $R$, we now require that for any $w \in R$, there is a set $S$ of non-overlapping substrings of $w$ which exhaust $w$. $S$ can be partitioned into two disjoint sets $S_r$ and $S_p$ of substrings of $w$ (the 'replacement regions' and 'preservation regions', respectively), such that:

1. every non-final $s \in S_p$ is directly followed by an $s' \in S_r$; (3)
2. every $s \in S_r$ is a replacement candidate:
3. every $s \in S_r$ matches $T$;
4. for every $s \in S_p$, there is no nonempty substring of $s$ which is a replacement candidate;
5. every $s \in S_p$ matches $\mathrm{id}(\Sigma)^*$.

Condition 1 states how we alternate between preserving non-candidates and replacing candidates. (Note that there is no need for a converse condition that every replacement region be followed by a preservation region, since the empty string, which follows any region, is a preservation region obeying conditions 4 and 5.) Condition 2 identifies replacement regions as such, and condition 3 ensures that the inputs of replacement regions are actually replaced. Condition 4 expresses the obligatoriness of the replacement, and condition 5 ensures that the inputs of preservation regions are actually preserved.

This informal definition can be translated into a formal one straightforwardly. Note that (3) makes third order quantification over sets of substrings (i.e. sets of sets of positions) which MSOL cannot express directly. However, substrings can be easily described as half-open intervals in terms of their boundary positions (in (4), the $x$s and $y$s of $B$ and the $z$s). We make use of the half-open interval version of *matches* discussed in section 3.3. The logic provides an interpretation of the position #, allowing any substring, even a suffix, to be specified as a half-open interval. The formula $\Phi_T$ corresponds to the unconditional replacement of $T$:

(4) $$\begin{aligned}\Phi_T \equiv_{\mathsf{def}} \exists B(\# \in B \wedge\\ &\textit{preserved-region}(0, \min(B)) \wedge\\ &\forall x, y(\textit{consecutive-in-set}(x, y, B) \rightarrow\\ &\quad \exists z(\textit{replaced-region}(x, z) \wedge\\ &\quad\quad \textit{preserved-region}(z, y))))\end{aligned}$$

The following define the derived predicates used in (4):

(5) $$\begin{aligned}&\textit{replaced-region}(x, y) \equiv_{\mathsf{def}}\\ &\quad \textit{replacment-candidate}(x, y) \wedge \textit{matches}(x, y, T)\end{aligned}$$

(6) $$\begin{aligned}&\textit{replacement-candidate}(x, y) \equiv_{\mathsf{def}} \qquad \text{(Unconditional version)}\\ &\quad \textit{matches}(x, y, \mathrm{dom}(T))\end{aligned}$$

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

(7) *preserved-region*$(x, y) \equiv_{\text{def}}$ (Version 1)
$\neg\exists z, w(x \leq z \wedge z < w \wedge z < y \wedge w \leq y \wedge \textit{replacment-candidate}(z, w))$
$\wedge\ \textit{matches}(x, y, \text{id}(\Sigma)^*)$

(8) *consecutive-in-set*$(x, y, S) \equiv_{\text{def}}$ (Version 1)
$x \in S \wedge y \in S \wedge x < y \wedge$
$\neg\exists z(z \in S \wedge x < z \wedge z < y)$

A pair of consecutive elements of $B$ form the boundaries of a sequence of a replacement region and a following preservation region. Together with the constraint that $[0, \min(S))$ is a preservation region, this expresses condition 1 of (6). The inclusion of # in $B$, also along with the clause allowing an initial preservation region, ensures that the regions are exhaustive. There is a transparent correspondence between the definition of *replaced-region* and conditions 2 and 3, and likewise between the definition of *preserved-region* and conditions 4 and 5.

### 6.2.2 Conditional replacement

We digress at this point to compare our approach to that of Karttunen (1997). He first defines an unconditional replacement operator that does not check context. In fact, the definition of unconditional replacement with extended regular expressions is very simple:

(9) $$(\text{id}(\text{no-upper}) \cdot T)^* \cdot \text{id}(\text{no-upper})$$

Here no-upper $= \neg(\Sigma^* \cdot (\text{dom}(T) - \epsilon) \cdot \Sigma^*)$, i.e. the set of all strings not containing $\text{dom}(T) - \epsilon$ as a substring. The correspondence to our informal definition is clear. Furthermore, our logical definition with boundaries can be seen to be equivalent to coding up Karttunen's version using the classical conversion from regular expressions to formulas (see Ebbinghaus and Flum (1999, Ch. 6)).

However, having defined unconditional replacement thus, it is necessary for Karttunen to resort to different techniques to place further conditions. The context check is achieved by using a transducer to mark up the input string with bracket symbols. Brackets are inserted around any replacement candidate, and then brackets are deleted when they don't occur in the right context (between $\lambda$ and $\rho$). Conditional replacement is then achieved by unconditionally replacing regions between brackets. Further constraints, like earliest and longest matching, involve using further constraints to filter out strings with certain bracket configurations.

Given the logical definition of unconditional replacement, all we need to do is to add a few clauses to it to get a conditional version. We simply reinterpret 'replacement-candidate' for $T/\lambda_\rho$ to mean a substring $s$ of a word $w$ such that $\text{dom}(s)$ matches $\text{dom}(T)$ *and* such that $s$ occurs in $w$ directly between an output $\lambda$ and an input $\rho$ (i.e. directly between substrings $l$ and $r$ s.t. $\text{ran}(l) = \lambda$ and $\text{dom}(r) = \rho$).

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

Formally:[7]

(10) $\mathit{replacement\text{-}candidate}(x, y) \equiv_{\mathsf{def}}$ (Conditional version)
$\mathit{matches}(x, y, \mathrm{dom}(T)) \wedge$
$\exists z(z \leq x \wedge \mathit{matches}(z, x, \Sigma^* \times\!\!\times \lambda)) \wedge$
$\exists w(w \geq y \wedge \mathit{matches}(y, w, \rho \times\!\!\times \Sigma^*))$

Using this conditional version of *replacement-candidate*, we will now write $\Phi_{T,\lambda,\rho}$ for (4). As we shall see in the next section, further restrictions are equally easy to add.

### *6.3 Directed replacement*

Just as conditional replacement can be derived from the logical definition of unconditional replacement with just a few extra clauses, directed replacement can be derived from the undirected version with little extra effort.

We focus on left-to-right (earliest match), longest match, rightward oriented replacement as a concrete example. We call this the EL-algorithm (for Earliest/Longest match). The 'left-to-right' property means that the choice of when to begin a replacement region is deterministic: the earliest possible one must always be chosen. The 'longest match' property also ensures that once a starting point for replacement is chosen, what exactly gets replaced is also deterministic: the longest candidate region must be chosen. Finally, the rightward orientation allows context checking to be done locally and incrementally in a left-to-right pass over the string. These three factors make it natural to state this kind of replacement as a deterministic[8] algorithm which makes a single left-to-right pass over the input string.

In figure 4 we present the algorithm in pseudocode. The parameters of the algorithm are the replacement relation $T$ and the context languages $\lambda$ and $\rho$. The input is given in an array `I`; output is written into a mutable length array `O`.[9]

### *6.4 Directed replacement: logical definition*

Now we extend $\Phi_{T,\lambda,\rho}$ to capture the replacement executed by the EL-algorithm. We can supplement our earlier informal declarative definition in (6) to require earliest matching by simply replacing condition 4 with the weaker 4′:

4′. for every $s \in S_p$, no nonempty substring of $w$ which is a replacement candidate begins within $s$;

[7] The same length cross-products with $\Sigma^*$ in (10) represent the fact that where output is being checked, the input can be any symbol, and vice versa.

[8] Strictly speaking, there may be a choice of more than one longest match replacement at a given point if the transducer specifying the replacement is non-functional, e.g. $a \times (b \cup c)$. The important point is that there is only one possible parse of the input string into replacement and preservation regions.

[9] We use the same conventions for arrays as for strings, including the half-open interval notation `I(i,j]`.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

```
Algorithm: earliest/longest match replacement T/λ__ρ
input: array I of length n
output: array O
i := 0; // index into I
o := 0; // index into O
while (i < n)
   // CASE 1
   if there exist j ≥ i, l ≤ i, r ≥ j such that
     (i)   I[i,j) ∈ dom(T), and
     (ii)  O[l,i) ∈ λ, and
     (iii) I[j,r) ∈ ρ
   then:
      // pick the longest replacement candidate:
      for the largest j s.t.
         I[i,j) ∈ dom(T) and
         I[j,r) ∈ ρ for some r ≥ j;
         // select a replacement for I[i,j):
         for w ∈ T(I[i,j)) with |w| = m
            // output the string w:
            O[o,o+m) := w;
            i := 1 + j;
            o := 1 + o + m;
  // CASE 2
   else:
      // preserve the ith input symbol
      O(o) = I(i);
      i := i+1;
      o := o+1;
end while
```

Fig. 4. EL-replacement algorithm.

In other words, you can't be preserving if you encounter an opportunity to start replacing. Similary, longest matching just corresponds to adding the condition 6:

6. for every $s \in S_r$, there is no substring $s'$ of $w$ which is a replacement candidate such that $s$ is a proper prefix of $s'$.

Formally, the weakened version of *preserved-region* and the strengthened version of *replaced-region* are derived from the earlier versions by subtracting and adding a clause, respectively:

(11) $\mathit{preserved\text{-}region}(x, y) \equiv_{\mathsf{def}}$ (Earliest match version)
$\neg\exists z, w(x \leq z \wedge z < w \wedge z < y \wedge \mathit{replacment\text{-}candidate}(z, w))$
$\wedge\ \mathit{matches}(x, y, \mathrm{id}(\Sigma)^*)$

(12) $\mathit{replaced\text{-}region}(x, y) \equiv_{\mathsf{def}}$ (Longest match version)
$\mathit{replacment\text{-}candidate}(x, y) \wedge$
$\mathit{matches}(x, y, T) \wedge$
$\neg\exists z(y < z \wedge \mathit{replacement\text{-}candidate}(x, z))$

When reinterpreted with these definitions for *preserved-region* and *replaced-region*, we write $\Phi_{T,\lambda,\rho}$ as $\Phi^{EL}_{T,\lambda,\rho}$

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

### 6.5 Correctness proof

We now give a semi-formal correctness proof, showing that the input-output relation described by the formula $\Phi^{EL}_{T,\lambda,\rho}$ is the same as that computed by the EL-algorithm. Recall that so far we are only dealing with the special case of same length replacements. What this means is that the input and output indices `i` and `o` in the algorithm will move in lockstep (i.e. `i=o` is a loop invariant), and the pair of input and output strings can be viewed as a string of pairs with a single index.

*Theorem*
For a same length relation $T$, for $s, t \in \Sigma^*$, $s \times t$ satisfies $\Phi^{EL}_{T,\lambda,\rho}$ iff the EL-algorithm maps $s$ to $t$.

*Proof*
*Part I*: If the EL-algorithm maps input $s$ of length $n$ to output $t$, then $s \times t$ satisfies $\Phi^{EL}_{T,\lambda,\rho}$.

Let # be $n$. Let $B'$ be the set of all values for `i` for which the algorithm enters the loop and activates case 1 and $B$ be $B' \cup \{n\}$.[10] Until the first replacement is made, the algorithm activates case 2 for each input symbol and outputs it unchanged. Since case 2 is only activated when case 1 fails, we know that no replacement candidate begins before $\min(B)$. Therefore, *preserved-region*$(0, \min(B))$ holds. Now take two consecutive members $x$ and $y$ of $B$. Since case 2 is activated at $x$, a replacement candidate begins at $x$ and the algorithm makes the replacement. By the longest match condition, we know that no longer replacement is possible. Let $z$ be the updated value of `i` after the replacement. Clearly *replacment-region*$(x, z)$ holds. Since $x$ and $y$ are consecutive in $B$, we know that case 1 is not activated again until $y$, so by reasoning parallel to that for the region $[0 \ldots \min(B))$, *preserved-region*$(z, y)$ holds.

*Part II*: if $s \times t$ of length $n$ satisfies $\Phi^{EL}_{T,\lambda,\rho}$, then the EL-algorithm maps $s$ to $t$.

In (i) below, we show that the output of the algorithm from `i := 0` to `i := min(B)` is $t[0, \min(B))$. Then in (ii) we prove in by induction over $B$ that for any consecutive $x, y$ in $B$, $t[x, y)$ is a possible output of the algorithm from `i := x` to `i := y`.[11] This entails that $t[\min(B), \#)$ is a possible output of the algorithm from `i := min(B)` to its termination (since case 1 nondeterministically selects a replacement of $s[x, z)$ for a fixed $x$ and $z$, the selections made each time case 1 is activated are independent of each other.) (i) and (ii) together verify the theorem.

(i) We know that *preserved-region*$(0, min(B))$ holds. So no replacement candidate starts before $\min(B)$, so the algorithm will activate case 2 for every position in that region. Since $\Phi^{EL}_{T,\lambda,\rho}$ requires that *matches*$(0, \min(B), \mathrm{id}(\Sigma)^*)$ and case 2 always outputs the input unchanged, the formula and algorithm agree for this region.

[10] For notational convenience, we conflate terms of the logic with their denotation, e.g. writing $B$ for the set of positions interpreting the variable $B$.

[11] If $T$ is functional, we can simply speak of *the* output; the only nondeterminism arises in case 1 in the choice of the replacement `w` from $T$(`I[i,j]`). Note in particular that `i` and `j` are fixed before the choice is made.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

(ii) *Base case*: For $B = \{n\}$, the theorem is proven. So assume $B$ is non-singleton and let $y$ be the second element of $B$. For some $z$, *replaced-region*$(\min(B), z)$ and *preserved-region*$(z, y)$ hold. We know either $\min(B) = 0$, or, by the reasoning of (i), case 2 was activated for `i := min(B) − 1`, so the algorithm enters the loop with `i := min(B)`. Since *replacement-candidate*$(\min(B), z)$ holds, and *replacement-candidate*$(\min(B), z')$ holds for no $z' > z$, case 1 is activated at `i := min(B)`, and some replacement is output for the region $[\min(B), z)$. We know that $t[x, z)$ is one of the possible outputs, since $s[x, z) \times t[x, z) \in T$ (because *replaced-region*$(\min(B), z)$ holds). Now, if $z = y$, the region $[z, y)$ is $\epsilon$, so assume $z < y$. Since *preserved-region*$(z, y)$ holds, the formula and algorithm agree for this region by the reasoning of part (i), and at `i := y − 1`, the algorithm activates case 2 and exits with `i := y`.

*Inductive step*: Let $x, y$ be consecutive in $B$, $x > \min(B)$ and assume that the algorithm activated case 2 at `i := x − 1` and exited with `i := x`. Since the algorithm enters the loop with `i := x`, then by reasoning parallel to that of the base case, it agrees with the formula for the region $[x, y)$. QED.

### 6.6 Non-same length relations

The operators defined so far use a logic of same length relations, so they cannot handle any replacements involving insertions or deletions, i.e. any $T/\lambda__\rho$ where $T$ is not a same length relation. However, this restriction can be easily circumvented by using a symbol `0` to mimic the empty string, as is done in two-level morphology (Koskenniemi 1983). Contractions are handled straightforwardly by mapping some input symbols to `0` and then composing the replace operator with a transducer that removes all `0`s from the output. For instance, to mimic $ab \rightarrow c/d__d$, we use $ab \rightarrow c0/d__d$. This is no problem, since $ab \times c0$ is a same length relation. The unneeded 0 is then removed by composition on the right with delete-0s:

$$\text{delete-0s} =_{\text{def}} (\text{id}(\Sigma) \cup (0 \times \epsilon))^* \tag{13}$$

At first glance, it seems like expansions could be handled the same way: compose the replace operator with a transduction insert-0s that inserts 0s anywhere into the input, i.e. the inverse of delete-0s. However, this will produce spurious outputs, since the transducer cannot generally know where to insert the 0s. Take the example of $a \rightarrow bc/d__d$, acting on the input string *dad*, where it should produce exactly one output, *dbcd*. Say we represent it as the same length relation $a0 \rightarrow bc/d__d$. This will do fine only for inputs like 000*da*0*d*0 in example (14a), which will be mapped to 000*dbcd*0 by the replace operator, and thence to *dbcd* by delete-0s. But other inputs will cause problems. The first problem is one of 'too few' zeros in the input. One of the results of inserting 0s into *dad* would be 00*dad*0 as in (14b). This does not contain *a*0 anywhere and will therefore undergo no replacement. The second problem is one of 'too many' zeros. Another result of inserting 0s into *dad* would be *d*0*a*0*d* as in (14c). This string does contain the correct input *a*0. However, although it is directly followed by a *d*, it is not actually directly preceded by a *d*; rather by *d*0,

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

which is not what the replacement asks for. Therefore, everything will be preserved again.

(14) a. $000da0d0 \rightarrow 000dbcd0 \rightarrow dbcd$
b. $00dad0 \rightarrow 00dad0 \rightarrow dad$
c. $d0a0d \rightarrow d0a0d \rightarrow dad$

The problem can be summed up this way. We know that we now intend the 0 symbol to be interpreted as null. However, the *matches* relation of the logic has not been informed of this change, and continues to treat it as just a normal symbol. The solution is thus to change *matches* to be sensitive to 0s.

What we would of course really like is a predicate *matches-weak* where *matches-weak*$(x, y, \mathscr{E})$ holds exactly when the interval $[x, y)$ matches some alignment of $\mathscr{E}$. However, this is not MSOL-expressible for arbitrary relations $\mathscr{E}$, as discussed in section 4.1. Specifically, we can only use such a predicate on the presupposition that $\mathscr{E}$ is recognizable.

However, for the replace operator, we do not need arbitrary weak matching. In the one case where we must require a substring to match an arbitrary relation, namely in the definition of *replaced-region*, literal matching to some alignment (of our choice) of the relation is all we need. Weakened matching is only needed on the input side (for the input check and the right context check) or the output side (for the left context check), but not both at once. We can define a version of *matches-weak*$(x, y, \mathscr{E})$ as in (15) which presupposes that $\mathscr{E}$ is a crossproduct. alignments-cp$(\mathscr{E})$ denotes alignments$(\mathscr{E})$ for cross-product relation $\mathscr{E}$ and is defined as in (16). It is necessary in (15) to apply $\gamma$ to $\mathscr{E}$ since $\mathscr{E}$ might in fact be a relation not over $\Sigma$ but over $\Sigma_0$, as when in (14), we must verify that the input matches $a0$ but only really care that it matches $a$.

(15) *matches-weak-cp*$(x, y, (\mathscr{E})) \equiv_{\text{def}}$
*matches*$(x, y,$ aligments-cp$(\gamma(\mathscr{E})))$

(16) alignments-cp$(\mathscr{E}) =_{\text{def}}$
ran(dom$(\mathscr{E}) \circ$ insert-0s) $\times\!\!\times$ ran(ran$(\mathscr{E}) \circ$ insert-0s)

alignments-cp inserts any number of 0s at any point in both the domain and the range of the original relation and then recombines the results as a same length relation.

The application of $\gamma$ to $\mathscr{E}$ solves the problem of 'too few' 0s. In example (14), instead of looking for inputs literally matching $a0$, we remove all 0s and look for inputs matching $a$ (and alignments-cp lets us also tolerate zero or more 0s on both sides of the $a$). The string $a$ in (14b) matches this. The 'two many' problem is solved by alignments-cp. Instead of looking for a left context with output $d$, we look for any string in $0^*d0^*$. The string $d0$ in (14c) matches this.

To update $\Phi_{T,\lambda,\rho}$ and $\Phi^{EL}_{T,\lambda,\rho}$ for non-same length relations, we replace *matches* with *matches-weak-cp* in the input and context checks in *replacement-candidate*, but leave *matches* in the definition of *replaced-region.* All occurrences of $\Sigma$ in the definitions

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

must also be replaced by $\Sigma_0$. The resulting weakened formulas will be written $\Phi^{\mathsf{weak}}_{T,\lambda,\rho}$ and $\Phi^{\mathsf{weak,EL}}_{T,\lambda,\rho}$

To implement the replacement $T/\lambda__\rho$ for arbitrary $T$, we first pick an arbitrary regular alignment $T'$ of $T$ (by constructing a transducer for $T$ and replacing $\epsilon$s with 0s as explained in section 4.1). We then use the relation insert-0s $\circ\ \Phi^{\mathsf{weak}}_{T',\lambda,\rho} \circ$ delete-0s (or with $\Phi^{\mathsf{weak,EL}}_{T',\lambda,\rho}$ as appropriate).

The correctness proof for the weakened version is somewhat complicated by the fact that in the EL-algorithm for a non-same length $T$, the input and output indices will generally be distinct at any point in the execution. However, the same proof principle can be used, since the input/output pair can still be aligned in such a way that it is partitioned into replacement and preservation regions.

### *6.7 Related work*

We briefly compare our declarative definition of the finite state replace operator with other versions in the literature.

Kaplan and Kay (1994) is the first paper to give a computationally implemented translation of replacement rules to transducers. Their method involves a cascade of transducers which insert brackets into the input string to mark off regions and then filter out incorrect configurations. In our logical definition, these brackets reappear as the first-order variables which demarcate the replacement and preservation regions, as well as those which demarcate rejected candidate regions in *preserved-region* and the earliest/longest match version of *replaced-region*. However, logical variables are a much simpler and cleaner method of marking boundaries. Kaplan and Kay use three different kind of variables which must be inserted, ignored while present in many cases where they are irrelevant, and then finally deleted. Their meaning is an emergent property of the cascade of transducers and is hard to isolate in a clear way that allows for rigorous verification. Logical variables over positions or regions on the other hand have a clear semantics.

All the other papers on the replacement operator cited use the same basic method of inserting and constraining brackets. Although subsequent versions such as Karttunen (1996, 1997) provide cleaner definitions and simplify the method by making use of fewer kinds of brackets, the same comment applies as to Kaplan and Kay (1994).

Mohri and Sproat (1996) provide a similar algorithm specified not in a regular expression calculus, but rather directly by operations on transducers. They are explicity concerned with verifiability and prove certain properties of some of the individual steps of their algorithm. However, they do not provide a rigorous proof of correctness.

Mohri and Sproat (1996) are also concerned with issues of efficiency and point out a problem with the algorithm of Kaplan and Kay (1994). For rewrite rules of the form $a \rightarrow b/__c^k$, Kaplan and Kay's algorithm takes time at least exponential in $k$. Since Mohri and Sproat identify certain complementation steps as the source of the problem, it might be feared that the logical definition would suffer the same problem because of the negations used, especially in the restrictions in

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

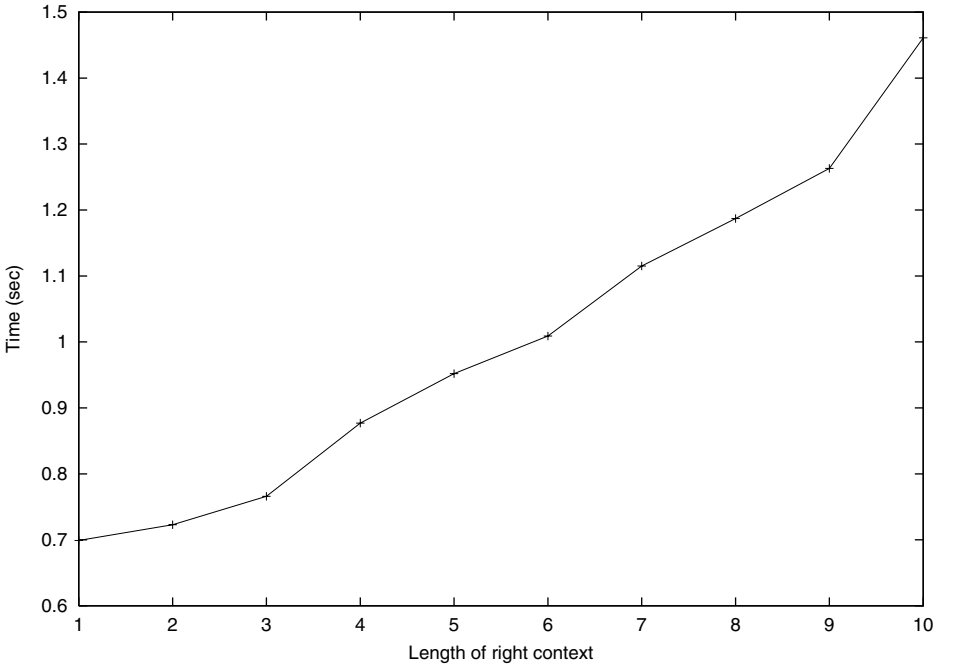



Fig. 5. Compilation time in seconds for $a \rightarrow b/__c^k$ as a function of $k$.

*replaced-region* and *preserved-region*. However, figure 5 shows that the time is roughly linear in $k$.[12]

Finally, Mohri and Sproat (1996) incorporate weighted replacements where different outputs of an input can be associated with different weights. It is interesting to consider whether such replacements can also be described logically. The simple generalization of same length relation to the weighted case is to relations where the weight of a string is a linear function of its length. It is possible to simulate other weight functions by again using 0s, having output symbols that carry weight but are ignored in the final output string. However, this still will only provide functions to rational values, not arbitrary real values. It remains to be investigated whether this restriction would yield a sufficiently general system in a sufficiently transparent way to make a logical description useful in this context.

## 7 Conclusion

This paper has demonstrated the usefulness of monadic second-order logic for describing regular relations. We have shown how the finite state replacement operator can be given a declarative definition in the logic, yielding a transparent proof of

[12] The vertical axis of figure 5 gives the compilation time in seconds on a Sparc Ultra-2 processor running Solaris 8 for the compilation in MONA of $a \rightarrow b/__c^k$ to an automaton, including the preprocessing step converting the specification of the regular expressions into definitions involving 16 bit-variables.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

correctness. We believe that this has demonstrated that MSOL is promising as a language for verifiable specification of finite state transductions and is a useful addition to the toolbox of NLP researchers and others dealing with finite state transduction.

## Acknowledgements

For valuable input at various stages of this work, thanks to Gosse Bouma, Chris Brew, Paul Davis, Jason Eisner, Dale Gerdemann, Erhard Hinrichs, Martin Jansche, Lauri Karttunen, Bob Kasper, Nils Klarlund, Rob Malouf, Detmar Meurers, Frank Morawietz, John Nerbonne, Gertjan van Noord, audiences at Tübinging, Groningen, and Ohio State University, participants in the FSMNLP workshop at ESSLLI '01 in Helsinki, and three anonymous reviewers. All shortcomings are mine.

## References

Berstel, J. (1979) *Transductions and Context-Free Languages.* Stuttgart: B. G. Teubner.

Büchi, J. R. (1960) Weak second order arithmetic and finite automata. *Zeitschr. f. math. Logik und Grundlagen d. Math* **6**: 66–92.

Chomsky, N. and Halle, M. (1968) *The Sound Pattern of English.* Harper and Row.

Ebbinghaus, H.-D. and Flum, J. (1999) *Finite Model Theory* (2nd ed.). Springer-Verlag.

Elgot, C. C. (1961) Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.* **98**: 21–52.

Gerdemann, D. and van Noord, G. (1999) Transducers from rewrite rules with backreferences. In *EACL 99*, Bergen, Norway.

Glenn, J. and Gasarch, W. (1997) Implementing WS1S via finite automata. *Automata Implementation, WIA'96: Lecture Notes in Computer Science 1260.* Springer-Verlag.

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

Johnson, D. C. (1972) *Formal Aspects of Phonological Description.* Mouton.

Kaplan, R. and Kay, M. (1994) Regular models of phonological rule systems. *Computational Linguistics* **20**(3): 331–379.

Karttunen, L. (1996) Directed replacement. *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, NM.

Karttunen, L. (1997) The replace operator. In: Roche, E. and Schabes, V., editors, *Finite-State Language Processing.* Bradford, MIT Press, pp. 117–147.

Kelb, P., Margaria, T., Mendler, M. and Gsottberger, C. (1997). MOSEL: A flexible toolset for MOnadic SEcond-order Logic. *Proc. TACAS'97, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems: Lecture Notes in Computer Science 1217*, pp. 183–202. Springer-Verlag.

Kempe, A. and Karttunen, L. (1996) Parallel replacement in finite state calculus. *COLING-96*, Copenhagen, Denmark.

Klarlund, N. (1999) A theory of restrictions for logics and automata. *Computer Aided Verification, CAV '99: Lecture Notes in Computer Science 1633.* Springer-Verlag.

Klarlund, N. and Møller, A. (2001) *MONA Version 1.4 User's Manual.* Department of Computer Science, Aarhus, Denmark. (Available from `http://www.brics.dk/mona/manual.html`.)

Klarlund, N. and Schwartzbach, M. I. (1999) A domain-specific language for regular sets of strings and trees. *IEEE Trans. Softw. Eng.* **25**(3): 378–386. (Extendend version of paper appearing in *USENIX Conference on Domain-Specific Languages*, Santa Barbara, 1997.)

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press

Koskenniemi, K. (1983) Two-level morphology: a general computational model for word-form recognition and production. Technical report 11, Department of General Linguistics, University of Helsinki, Finland.

Meyer, A. R. (1975) Weak monadic second-order theory of successor is not elementary recursive. In: Parikh, R., editor, *Logic Colloquium (Proc. Symposium on Logic, Boston 1972): Lecture Notes in Computer Science 453*, pp. 132–154. Springer-Verlag.

Mohri, M. and Sproat, R. (1996) An efficient compiler for weighted rewrite rules. *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, NM.

Roche, E. and Schabes, V. (editors) (1997) *Finite-State Language Processing*. MIT Press.

Thomas, W. (1996) Languages, automata, and logic. Technical report, Bericht 9607, Institut für Informatik und Praktische Mathematik, Universität Kiel.

van Noord, G. (n.d.) *FSA6 Reference Manual*. Alfa-informatica, University of Groningen. (Available from `http://odur.let.rug.nl/~vannoord/Fsa/Manual/`.)

van Noord, G. and Gerdemann, D. (1999) An extendible regular expression compiler for finite-state approaches in natural language processing. *Workshop on Implementing Automata 99*.

https://doi.org/10.1017/S1351324903003103 Published online by Cambridge University Press