Mix and Match Replacement Rules

Dale Gerdemann
Department of Linguistics
Universität Tübingen
72074 Tübingen, Germany
dg@sfs.uni-tuebingen.de

Abstract

A flexible construction kit is presented compiling various forms of finite state replacement rules. The approach is simpler and more declarative than algorithms in the tradition of Kaplan & Kay. Simple constraints can be combined to achieve complex effects, including effects based on Optimality Theory.

1 Introduction

Traditional finite state algorithms for compiling replacement rules have a very procedural flavor. The general approach originated by Kaplan & Kay [8] involves steps for introducing markers, constraining the markers in various ways, making the actual replacement, constraining the markers again after the replacement and finally removing the markers. Several variants of Kaplan & Kay's algorithm have been presented ([9], [12], [4]) with the goals being either to improve efficiency or to provide slightly different semantics. In this paper, I present an alternative, more declarative approach. Rather than being based on a sequence of steps that are composed together, this approach is based on intersection. A basic, unconstrained replacement is represented by a transducer, which can then be constrained by intersecting this transducer with automata representing a variety of constraints. The obvious objection to this approach is that transducers are not closed under intersection.[1] It turns out, however, that a rather limited form of intersection is sufficient for implementing the various forms of replacement rules that have appeared in the literature, along with a wide variety of replacement rules that have not appeared in the literature.

One of the advantages of finite state technology is reusability. An implementation designed for one finite state toolbox can easily be redeployed with a different toolbox. At least that is the theory. In practice, toolboxes tend to provide a number of incompatible features. Replacement rules, in particular, occur in many variations: optional/obligatory, left-to-right/right-to-left/simultaneous, longest-matching/shortest-matching, etc. Sometimes a rather small variation has required introducing some new syntax. The widely used Xfst system [1], for example, has special syntax (dotted brackets) used to enforce a constraint against repeated epenthesis. While this is certainly a useful option, it is also an impediment to users who wish to transfer their Xfst implementation to another toolbox. Clearly such users would benefit more from a kind of replacement rule construction kit, allowing such variant replacement rules to be put together as needed. But the procedural style of Kaplan & Kay-inspired algorithms makes the development of such a construction kit very difficult. Given the declarative approach developed in this paper, the construction kit approach can be implemented in any standard finite-state toolkit.

The declarative approach to replacement rules is very flexible, allowing for example, variants of finite-state Optimality Theory ([2] [5]) to be incorporated into the rules. In some sense, this is not new, as Karttunen's approach to longest match replacement [9], was already a kind of optimization. The declarative, mix-and-match approach, however, allows for the easy incorporation of various optimality constraints. For example, a constraint against repeated epenthesis could be loosened to allow the minimal amount of epenthesis consistent with some other constraints.[2]

2 Flat Representation for Transducers using Types

The general form for a finite-state rewrite rule is as in 1:

$$x \rightarrow T(x)/\lambda _ \rho \quad (1)$$

This is understood as a rule for replacing the string x with the transduction $T(x)$ in the specified context, with some unspecified mode of operation such as obligatory left-to-right. Traditionally T is specified as a cross-product of regular languages, but as discussed in Gerdemann & van Noord [4], this is by no means necessary.

In order to develop an approach based on intersection, a rule such as 1 (ignoring context, and other constraints) will first be transformed into a finite state acceptor (FSA). There is of course a standard trick of using same-length transducers, which goes back to the early days of finite-state morphology [11]. A same-length transducer can be encoded as an FSA with

[1] More precisely, regular relations are not closed under intersection. It is convenient, however, to systematically confuse finite state transducers (respectively, finite state acceptors) with the regular relations (respectively, regular languages) that they encode.

[2] This paper concentrates on some fairly simple examples that are good for expository purposes. For further examples, see: www.sfs.uni-tuebingen.de/∼dg/mixmatch.

Workshop Adaptation of Language Resources and Technology to New Domains 2009 - Borovets, Bulgaria, pages 39–47

transitions labeled with pairs of symbols, or equivalently by an FSA in which odd-numbered transitions (from the start state) count as inputs, and even numbered transitions count as outputs. This is, however, an inconvenient representation for two reasons. First, we often work with transducers that do not have the same-length property. And second, we would like our flattened transducer representations to contain other symbols for markup which count as neither input nor output. So rather than force things into a same-length mold, we use the following very general definition for a flat representation of a transducer:[3]

Definition 2.1 *A finite state automaton **A** is a **flat representation** of a transducer **T** with respect to transducers **extract_input** and **extract_output** if:*

$$T = \mathit{extract_input}^{-1} \circ \mathit{identity}(A) \ \circ \mathit{extract_output}$$

It should be understood here that the equals sign means that the transducers defined on either side represent the same regular relation,

This flat representation is the central feature of the approach. This approach allows *extract_input* and *extract_output* to be defined in many different ways. To instantiate these extraction transducers in a useful and flexible way, I start by introducing a type system with types that these transducers can refer to.

2.1 Types

The goal of introducing types is to make it easy to define *extract_input* and *extract_output*. If some symbols are of the input type, for example, then these are the ones that should be extracted by *extract_input*. A simple idea would be to use different alphabets for these two types. In some cases, this is a convenient idea. For example, if the input alphabet is Cyrillic and the output alphabet is Latin, then it is easy to distinguish input and output symbols. In most cases, however, this is not practical. Therefore an alternative approach is adopted: every untyped symbol is followed by a *symbol type indicator*, which specifies the type of its predecessor.

At a minimum, there need to be two symbol type indicators: one for input symbols and one for output symbols. More usefully, there should also be indicators for symbols that are neither input nor output (marker symbols) and also for symbols that are both input and output (identity symbols). Possibly, for some algorithms, it would be useful to have a several types of marker symbols, but for now four symbols will be enough.

- define input a;
- define output b;
- define identity c;
- define marker x;
- define symbolTypeIndicator
  [input ∪ output ∪ identity ∪ marker];

[3] It is nevertheless possible using a two-level approach to work out ideas similar to those in this paper. See Yli-Jyrä [15].

Note that the self-explanatory syntax here is as in the Foma system (Hulden [6]). The particular choice of symbols used here is arbitrary and can easily be changed,

Extended alphabet symbols thus consist of a sequence of two symbols, where the second symbol determines the type of the first symbol. With the preceding conventions, for example, the sequence [z a] (= [z input]) represents ‘z’ as an input symbol, and [< x] (= [< marker]) represents the angle bracket ‘<’ used as a marker symbol. For the purpose of illustration with a running example, a small test alphabet is defined here (even though Foma does not require the alphabet to be specified). In these definitions, *xsig* should be understood as “extended sigma.”

- define sigma [a ∪ b ∪ c ∪ x]; # small test alphabet
- define xsig [sigma symbolTypeIndicator];

An approach using sequences of two symbols as the basic unit can be awkward to work with and painful to debug. Thus it is essential to provide a set of tools that are specialized to work with this approach.

2.1.1 Matchers, Constructors and Accessors

Three basic tools for working with an extended alphabet are *matchers*, *constructors* and *accessors*. Generally, these tools are designed to hide the symbol type indicators from the user.

Matchers Matchers are a set of defined regular expressions for matching the various types of symbols.

- define in [sigma [input ∪ identity]];
- define inx [sigma input];
- define out [sigma [output ∪ identity]];
- define outx [sigma output];
- define ident [sigma identity];

Note that the ‘x’ in *inx* and *outx* should be read as “except identity.” Also note that these definitions use the toy alphabet *sigma*. For general use, one should replace this with Σ, which is the Foma symbol for an open alphabet.

Constructors Constructors are used to turn an ordinary alphabet symbols into typed symbols of the extended alphabet. For this purpose, parameterized definitions are used.

- define in(x) [x [input ∪ identity]];
- define inx(x) [x input];
- define out(x) [x [output ∪ identity]];
- define outx(x) [x output];
- define ident(x) [x [identity]];

Accessors Accessors are used to extract either the input side or the output side. By extracting both sides, a definition can be given for *unflatten*. The definitions rely upon a definition of term complement which is specialized for use with the extended alphabet.[4] The first step in *extractInput* eliminates all non-input symbols. This is composed with a transducer that eliminates the symbol type indicators.

- define tcomp(e) xsig-e; # term complement
- define extractInput
  [tcomp(in):ϵ $\cup$ in]* $\circ$ [Σ Σ:ϵ]*;
- define extractOutput
  [tcomp(out):ϵ $\cup$ out]* $\circ$ [Σ Σ:ϵ]*;
- define unflatten(ϕ)
  extractInput^{-1} $\circ$ ϕ $\circ$ extractOutput;

2.1.2 Pretty Printers

Just as in any programming language, it is important to have concise, understandable print representations of complex data structures. Most finite state toolboxes include a graphical interface, which is very helpful for debugging. It is certainly helpful to have such an interface, but experience has shown that networks quickly become way too complicated to be understood visually. Effective debugging is an acquired skill that requires a great deal of ingenuity in breaking problems into units that can be understood as networks. Since the approach of using a typed alphabet doubles the size of the network, it must also at least double the cognitive complexity of understanding the network.[5]

What is needed is a graphical interface that shows the types of symbols in a concise way without using sequences of two symbols, and without using the symbol type indicator. An implementation might, for example, use different colors for the different types. But since such an implementation does not exist, I simply define here an ad hoc pretty printing approach, specialized for use with the small toy alphabet. The idea is that an input 'a' is transduced to 'a:', an output 'a' is transduced to ':a', and an identity 'a' is transduced to 'a'. In all cases, the symbol type indicator is removed.[6]

- define ppInput
  [inx(a):a%: $\cup$ inx(b):b%: $\cup$ inx(c):c%:
  $\cup$ inx(x):x%:];
- define ppOutput
  [outx(a):%:a $\cup$ outx(b):%:b $\cup$ outx(c):%:c
  $\cup$ outx(x):%:x];
- define ppIdentity
  [ident(a):a $\cup$ ident(b):b $\cup$ ident(c):c
  $\cup$ ident(x):x];
- ppIO [ppInput $\cup$ ppOutput $\cup$ ppIdentity];
- define pp(X) [X $\circ$ ppIO*]$_2$;

Further utilities and pretty printers will be introduced as needed. First, however, we should start to see how the type system can be useful for our compilation.

2.1.3 Assertions and Boolean Tests

An important invariant of the typed approach is that every sequence is of even length and every symbol in an even position is a symbol type indicator. Consistent use of matchers, constructors and accessors will help to ensure that this invariant holds. Nevertheless, things can go wrong, so it is important to use boolean tests as assertions. Boolean types were introduced into the regular expression calculus in van Noord & Gerdemann [14], though the details are worked out differently here. The central idea is similar to conventions in a variety of programming languages, where certain designated values are understood as, or coerced to boolean values. In the finite state domain, it is convenient let ϵ be `true`, and $\emptyset$ be `false`.

- define true ϵ;
- define false $\emptyset$;

Note that with these conventions, `true` is a single-state FSA where the one state is both initial and final, and `false` is a single-state FSA where the one state is non-final.[7]

Given these conventions, the boolean connectives *and*, *or* and *not* can be defined as concatenation, union and complementation, respectively.

- and(B1, B2) B1 B2;
- or(B1, B2) B1 $\cup$ B2;
- not(B) $\neg$B $\cap$ true;

---

[4] The expression *tcomp(in):*ϵ $\cup$ *in* could be written more explicitly as *tcomp(in):*ϵ $\cup$ *identity(in)*. It is, however, standard in the literature to let language expressions be coerced to identity relation expressions when the context requires this interpretation.

[5] Besides raising the cognitive complexity, the typed approach also clearly increases the computational complexity. But this is unlikely to be a practical concern. The complexity of finite state implementations does not, in any case, derive from the complexity of individual replacement rules. Normally, complexity derives from composing a long sequence of such replacement rules.

[6] Note that the percent sign is used in Foma to escape special characters. So 'a%:' is the two-character symbol consisting of an 'a' followed by a colon. Note also that the subscript 2 in the last definition is used to extract the second projection (also known as "range" or "lower language").

[7] In Foma, it is easy to see if assertions have succeeded or not from the compiler output. For example, with a source file (assert.fom) containing the following two lines

```
define assertion1 ε;
define assertion2 ∅;
```

the compiler produces the following output

```
foma[0]: source assert.fom
Opening file 'assert.fom'.
defined assertion1: 136 bytes. 1 states, 0 arcs, 1 path.
defined assertion2: 136 bytes. 1 states, 0 arcs, 0 paths.
```

The information that the compiled version of *assertion1* has 1 path is sufficient to see that this assertion succeded.

For an example using a lot of assertions, see www.sfs.uni-tuebingen.de/~dg/mixmatch/FinnOTMatching.fom. The FinnOTMatching program is also a good illustration of finite state OT in general.

With these connectives, basic decidable properties of FSA's can be defined.[8]

- empty(L) not([L:true]$_2$);
- subset(A1,A2) empty(A1 ∩ ¬A2);
- equal(A1, A2)
  and(subset(A1,A2), subset(A2,A1));

Using these tools, the following assertions can be defined.

- define evenLength(L) empty(L ∩ [[Σ Σ]* Σ]]);
- define indicatorsInEvenPositions(L)
  subset([[[Σ Σ]* Σ]:ϵ Σ Σ*:ϵ]$_2$,
  symbolTypeIndicator);

3 Basic Unconstrained Replacement

To compile a replacement rule as in (1), we must start by flattening the transducer T. We have already seen the *unflatten* definition, and it is clear that *flatten* should be the inverse of this, so that for transducer T:

$$T \equiv unflatten(flatten(T)) \qquad (2)$$

If T is a cross-product transducer, then this is easy to define:

- define flattenCross(ϕ, ψ)
  [$\phi \circ$ [Σ ϵ:input]*]$_2$ [$\psi \circ$ [Σ ϵ:output]*]$_2$;

Flattening for non-cross product transducers, although not hard, requires access to states and transitions and therefore cannot be done in a portable way.[9]

In general, the flattening step may arbitrarily intersperse input and output symbols, making it impossible to write constraints that refer at the same time to input and output symbols. The general problem is that transducers are not closed under intersection. The good news is that constraints on replacement rules rarely need to refer simultaneously to inputs and outputs.

Given a flattened transducer T, we can define a flattened unconstrained replacement rule with T as its center. By analogy with Optimality Theory, this basic replacement rule is called *replaceGen*.

- define lb [a marker];
- define rb [b marker];
- define bracket lb ∪ rb;
- define replaceGen(ϕ) [ident ∪ [lb ϕ rb]]*;

Since these rules have introduced a couple of marker symbols, it is important to define corresponding pretty printing definitions.

- define ppBrackets [lb:%< ∪ rb:%>];
- define pp1(X) [X $\circ$ [ppIO ∪ ppBrackets]*]$_2$;

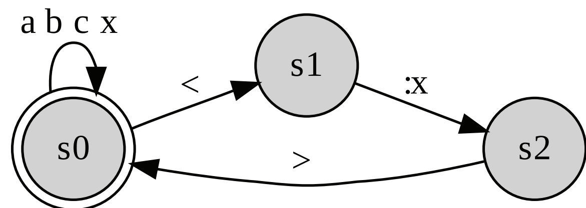

Fig. 1: *regex pp1(replaceGen(flattenCross(ϵ, x)))*

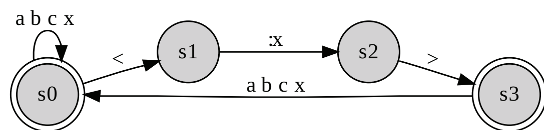

Fig. 2: *Non-iterated Epenthesis, flattened version*

Putting these pieces together, we see in Figure 1 the pretty-printed flattened automaton for free epenthesis of 'x'. This automaton looks superficially like an intermediate step in a traditional replacement rule compilation procedure. But in fact, it is considerably different. The brackets, for example, are not inserted as steps in a procedure. Constraints may refer to the brackets, but the brackets are neither in the input nor in the output.

4 Constraints

Constraints are of two basic types: strict and optimizing. Contextual constraints are, for example, traditionally treated as strict constraints, and optimizing constraints are typically limited to constraints for leftmost and longest matching (Karttunen [9]). Strict constraints, being easier, are a natural starting point.

4.1 Strict Constraints

As a simple example of a strict constraint, consider a constraint against iterated epenthesis. This may seem like a minor constraint, but nevertheless it was considered important enough to merit some special syntax in Xfst.[10] This is easy to define in the mix-and-match approach, using auxiliary definitions for complement and containment. The automaton is shown in Figure 2.

- define comp(E) xsig* - E;
- define contain(E) [xsig* E xsig*];
- define noIterEpenStrict
  comp(contain([lb outx* rb lb outx* rb]));
- regex pp1(replaceGen(flattenCross(ϵ, x))
  ∩
  noIterEpenStrict);

[8] For transducers, Foma provides the boolean tests: *_isfunctional* and *_isidentity*.

[9] See: www.sfs.uni-tuebingen.de/∼dg/mixmatch/flatten for a mix of Foma code and Perl code.

[10] See Beesley & Karttunen [1], p. 67.

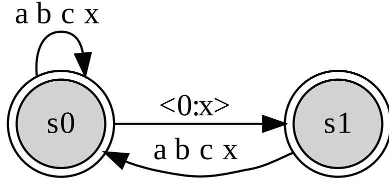

Fig. 3: *Non-iterated Epenthesis, unflattened version*

The unflattened version, compiled from the following code, is shown in Figure 3.

- regex unflatten(
  replaceGen(flattenCross(ϵ, x))
  $\cap$
  noIterEpenStrict);

5 Defeasible Constraints

Now let us relax the constraint against iterated epenthesis and allow iterated epenthesis only when there is no other alternative. Suppose that there is a higher ranking constraint requiring the output to have a sequence of two x's. If the input already has a sequence of two x's, then the output constraint will in any case be satisfied. If the input only has singleton x's, then epenthesis is needed. But iterated epenthesis is not needed. If, however, the input has no x's, then iterated epenthesis is needed. But it is still desirable to do no more iterated epenthesis than necessary. We start, as in OT, by putting a star after every constraint violation. To do this, we need some standard utilities from Kaplan & Kay [8].[11]

- define intro(S) [[Σ Σ] ∪ ϵ:S]*;
- define introx(S) [[Σ Σ] ∪ ϵ:S]* [Σ Σ];
- define xintro(S) [Σ Σ] [[Σ Σ] ∪ ϵ:S]*;

- define ign(L, S) [L ∘ intro(S)]$_2$;
- define ignx(L, S) [L ∘ introx(S)]$_2$;
- define xign(L, S) [L ∘ xintro(S)]$_2$;
- define ifPThenS(P,S) comp([P comp(S)]);
- define ifSThenP(P,S) comp([comp(P) S]);
- define PIffS(P,S) ifPThenS(P,S) ∩ ifSThenP(P,S);

Using these definitions, we can introduce a star, as in OT, and constrain it to occur after constraint violations.

- define star [c marker];
- define ppStar star:%*;

[11] As a useful convention, 'x' at the beginning means "except at the beginning," and similarly for 'x' at the end. The abbreviation *ign* is used for ignore.

- define pp2(X)
  [X ∘ [ppIO ∪ ppBrackets ∪ ppStar]*]$_2$;
- define markViolation(τ, π)
  ign(τ, star)
  $\cap$
  PIffS([xsig* ignx(π,star)], [star xsig*]);

Then we can mark the violation. The result is shown in Figure 4.

- regex pp2(markViolation(
  replaceGen(flattenCross(ϵ, x)),
  [lb outx* rb lb outx* rb]));

The next step involves minimizing the number of stars. We want to rule out the case that an input would be mapped to different outputs, b and w (for "better" and "worse"), where output b contains n stars, output w contains m stars, and $m > n$. The general idea is to construct a *worsening transducer* that turns better candidates into worse ones. Then, given a set of candidates c, the optimal candidates will be $c - worsen(c)$. This assumes, of course, that the worsening of a candidate is complete, all worse candidates are included and no non-worse candidates are included. In some cases, this relation can only be approximated.

There are two kinds of worsening: star-based worsening, which is the main focus of this paper, and generalized worsening. The difference is that star-based worsening works by adding stars to a candidate, whereas generalized worsening directly manipulates the output for a given input to make it worse. In either case, worsening is used to filter out bad candidates by comparison with alternative candidates for the same input. This means that GEN cannot actually change the input, GEN can only add *markup symbols* into the input, where markup symbols are disjoint from input symbols.[12]

For *replaceGen*, the obvious markup symbols are the brackets and the output symbols. But what about the identity symbols? Here, an input symbol $[\Sigma, input] = [\Sigma, a]$ is "marked up" as $[\Sigma, identity] = [\Sigma, c]$. To deal with this problem, the *changeMarkup* step is written to allow identity symbols into non-identity input symbols, and vice versa.

- define changeIdentities
  [[Σ identity:input] ∪ [Σ input:identity] ∪ xsig]*;
- define changeOutputs
  [outx:ϵ ∪ ϵ:outx ∪ xsig]*;
- define changeBrackets
  [bracket:ϵ ∪ ϵ:bracket ∪ xsig]*;
- define changeMarkup(X) [
  X
  ∘ changeIdentities
  ∘ changeOutputs
  ∘ changeBrackets]$_2$;

[12] So GEN can only change the input indirectly by adding markup symbols that are interpreted as editing instructions. For example, if GEN brackets syllables, then unsyllabified parts of the input may be deleted in a later (phonetic) module. Other kinds of editing interpretations for markup can easily be invented.

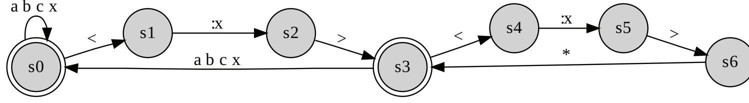

Fig. 4: *No iterated Epenthesis with violations starred*

Returning now to our running example, suppose that we add a higher ranked constraint which in some cases can only be satisfied by use of iterated epenthesis. Specifically, let us require that the output must contain a sequence of two instances of 'x'. Since this is a strict constraint, it can be applied by intersection. The result is shown in Figure 5.

- define containsOutputXXGen
  replaceGen(flattenCross(ϵ, x))
  $\cap$
  contain([out(x) tcomp(out(Σ))* out(x)]);

We then mark violations of the no iterated epenthesis constraint, with the result shown in Figure 6.

- regex pp2(markViolation(
  containsOutputXXGen,
  [lb outx* rb lb outx* rb]));

The automaton in Figure 6 is fairly complicated. After changing markup, however, it is greatly simplified, since constraints on correct ordering of output are abstracted away. The result is shown in Figure 7.

- regex pp2(changeMarkup(markViolation(
  containsOutputXXGen,
  [lb outx* rb lb outx* rb])));

Figure 7 shows that every candidate has one of three things.

1. an identity x, or
2. an output-only x, or
3. a star.

If we now add in a positive number of stars, we obtain an automaton that matches every non-optimal candidate as seen in Figure 8.

- regex
  pp2(addStars(changeMarkup(markViolation(
  containsOutputXXGen,
  [lb outx* rb lb outx* rb]))));

In general, however, just adding stars is not sufficient to obtain an automaton that matches every non-optimal candidate. A simple example suffices to illustrate the problem. Suppose that there are two candidates, where the better one is *ab*c* and the worse one is *a*bc**. Here it is not possible to add a star to the better candidate to match the worse candidate. The issue is that stars need to be not only added but also moved around in order to match up. Here it is in general not possible using finite state methods, to obtain all possible placements of the stars. So various levels of approximation must be defined.

- define permuteStarLeft
  [ϵ:star tcomp(star)* star:ϵ];
- define permuteStarRight
  [star:ϵ tcomp(star)* ϵ:star];
- define permuteStars
  [xsig* [permuteLeft $\cup$ permuteRight]]* xsig*];
- define permute0(X) X_2;
- define permute1(X) $[X \circ \text{permuteStars}]_2$;
- define permute2(X)
  $[X \circ \text{permuteStars} \circ \text{permuteStars}]_2$;
- etc

In theory, an unbounded amount of permutation may be necessary in order to optimize all possible inputs. In practice, however, this is only the case in highly artificial examples such as the one constructed by Frank and Satta [2]. The normal case is that only *permute0* and *permute1* are needed. To test how much permutation is needed, one can use the *isFunctional*[13] test described in Gerdemann & van Noord [5].

- define isOptimized(Starred)
  _isfunctional([Unflatten(Starred)
  $\circ$
  Σ-star -> ϵ]);

The idea here can be illustrated quite simply. Suppose the input *abc* results in two outputs *xa*ybc** and *axb*z* where $\{x, y, z\}$ is markup. If we delete everything but the stars, then *abc* is mapped to **** and ***. The *isFunctional* test is designed to rule out cases like this. Each input should be mapped to a set of candidates with a unique number of stars. If this condition holds, then the set of candidates has been optimized.

By using the *isFunctional* test, it can be determined that no permutation of stars is necessary for this case. So we can optimize in the following way, where the result is shown in Figure 9.

- define optimizeStars0(StarredCandidates) [
  StarredCandidates
  -
  permute0(
  addStars(
  changeMarkup(StarredCandidates)))];

[13] In Foma, this is called _isfunctional. The return values are `true` and `false` as described in section 2.1.3,

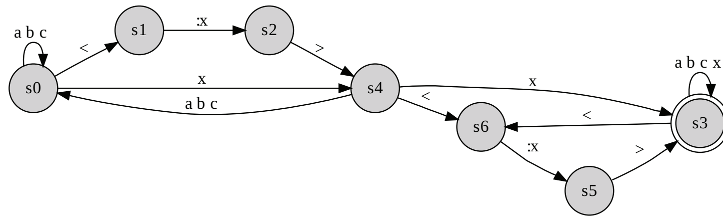

Fig. 5: *containsOutputXXGen*

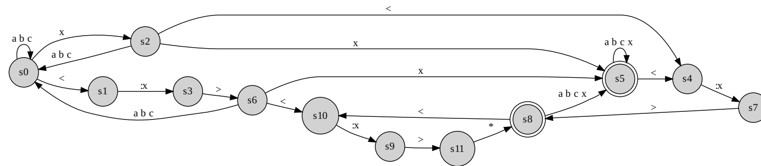

Fig. 6: *Starred violations of no iterated epenthesis*

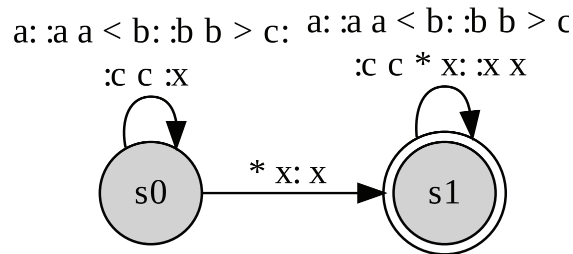

Fig. 7: *After changing the markup*

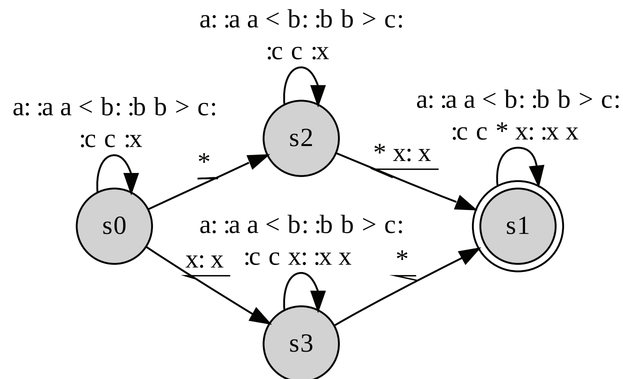

Fig. 8: *After adding stars*

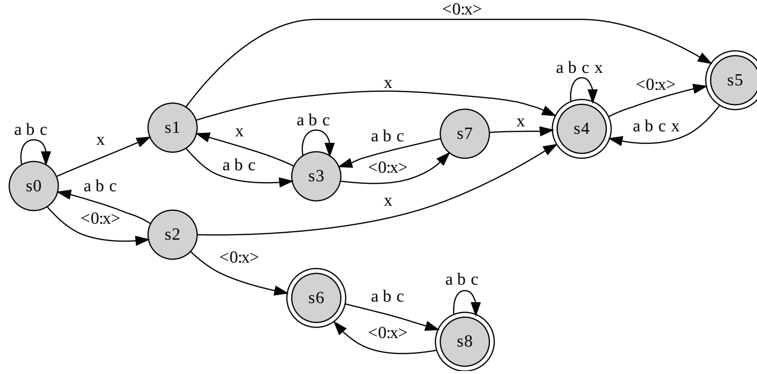

Fig. 9: *After optimizing*

- define opimizedRewriteRule
  unflatten(
  optimizeStars0(
  markViolation(
  containsOutputXXGen,
  [lb outx* rb lb outx* rb])));

The transducer in Figure 9 is fairly complex, so it is not obvious that it is correct. The skeptical reader should study some sample input-output such as the following.

- ab ↦ {abxx,axbxx,axxb,axxbx,xabxx,xaxbxx, xaxxb,xaxxbx,xxab,xxabx,xxaxb,xxaxbx}
- ax ↦ {axx,axxx,xaxx,xaxxx}
- axx ↦ {axx,axxx,axxxx,axxxxx,xaxx,xaxxx, xaxxxx,xaxxxxx}

6 Conclusion and further directions

It may seem like a lot of work has been expended in order to define a fairly simple replacement rule. But most of this work has gone into preparatory ground work. Further constraints can now be added in a fairly routine way. Certainly, constraints referring to the left and right context are necessary, and now are easy to define. And other, more exotic constraints are not difficult. The user of a toolbox is thus not bound by the particular flavors of replacement rules provided by the toolbox.

For optimizing constraints, I have introduced two kinds of worsening: star-based and generalized. Though this paper has concentrated on star-based worsening, it may well be the case that generalized worsening is even more important. The idea of generalized worsening is that markup can be directly manipulated to make candidates worse. For example, for a longest match constraint, the worsening transducer could manipulate the marked-up matches to make the matches shorter. Or a worsener for a leftmost constraint could manipulate markup to move matches further to the right. It is clear that a worsening transducer should encode an irreflexive, asymmetrical, transitive relation. Although these properties are in general undecidable [7], in practical cases it is usually clear enough that a worsening transducer is well formed.

Generalized worsening is particularly useful for describing prosodic constraints. For example, medieval Indian prosodists introduced following hierarchy of long-short syllable patterns (Singh, [13]), where long syllables count as two positions and short syllables count as one: SSSSS < LSSS < SLSS < SSLS < LLS < SSSL < LSL < SLL.[14] This looks remarkably like some versions of generalized alignment in Optimality Theory. For star-based worsening, the relevant constraint assigns stars to each long syllable, where the number of stars for a long syllable starting in position i is F_{i+1}, the $i+1^{st}$ Fibonacci number, and short syllables do not get stars. So with stars added, the hierarchy becomes: SSSSS < L*SSS < SL**SS < SSL***S < L*L***S < SSSL***** < L*SL***** < SL**L*****. Clearly it is not possible to define a transducer to add such a pattern of stars, so the only hope here is to use generalized worsening. The details are a little tricky, but the basic principle is to treat L as a bracketed pair of S's and let worsening move the bracketing to the right or introduce additional bracketing.[15]

Although there may be tricky issues in defining a generalized worsening transducer, the optimization itself is very straightforward.

- define optimize(G, Worsener)
  G - [G ∘ Worsener]$_2$;

To summarize, then, replacement rules have been encoded in a declarative, one-level way, allowing both optimizing and non-optimizing constraints to be mixed and combined in a natural and easy way. In combining OT-style constraints with traditional Generative Phonology-style replacement rules, one may see Kisseberth [10] as a source of inspiration. Long ago, before OT was invented, Kisseberth spoke of rules conspiring to create certain surface effects. The approach presented in this paper is intended to allow such conspiratorial effects to be compiled directly into the replacement rules.

[14] For some issues concerning this pattern, see Gerdemann [3].
[15] See www.sfs.uni-tuebingen.de/∼dg/mixmatch/zeck.fom for details.

References

[1] K. R. Beesley and L. Karttunen. *Finite State Morphology*. CSLI, Stanford, 2003.

[2] R. Frank and G. Satta. Optimality theory and the computational complexity of constraint violability. *Computational Linguistics*, 24:307–315, 1998.

[3] D. Gerdemann. Combinatorial proofs of Zeckendorf family identities. *Fibonacci Quarterly*, pages 249–262, August, 2008/2009.

[4] D. Gerdemann and G. van Noord. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway, 1999.

[5] D. Gerdemann and G. van Noord. Approximation and exactness in finite state optimality theory. In J. Eisner, L. Karttunen, and A. Thériault, editors, *SIGPHON 2000, Finite State Phonology. Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*, Luxembourg, 2000.

[6] M. Hulden. Foma: a finite-state compiler and library. In *Proceedings of the EACL Demonstrations Session*, pages 29–32, 2009.

[7] J. H. Johnson. Rational equivalence relations. *Theoretical Computer Science*, 47(1):39–60, 1986.

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

[9] L. Karttunen. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[10] C. Kisseberth. On the functional unity of phonological rules. *Linguistic Inquiry*, 1:291–306, 1970.

[11] K. Koskenniemi. Two level morphology: A general computational model for word-form recognition and production, 1983. Publication No. 11, Department of General Linguistics, University of Helsinki.

[12] M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[13] P. Singh. The so-called Fibonacci numbers in ancient and medieval India. *Historia Mathematica*, 12(3):229–244, 1985.

[14] G. van Noord and D. Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam, Germany, 1999.

[15] A. Yli-Jyrä. Transducers from parallel replace rules and modes with generalized lenient composition. In *Finite-state methods and natural language processing*, 2007.