SIGPHON 2000

Finite-State Phonology

Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology

Jason Eisner, Lauri Karttunen and Alain Thériault
Editors

6th August 2000

Centre Universitaire

Luxembourg

COLING 2000

© 2000 by the respective authors.

Additional copies of these proceedings may be ordered while stocks last from :

Association for Computation Linguistics (ACL)
75 Paterson Street, Suite 9
New Brunswick, NJ 08901 USA
Tel : +1-732-342-9100
Fax : +1-732-342-9339
Email: **acl@aclweb.org**

An online version of this proceedings is available at
the SIGPHON web site:

http://www.cogsci.ed.ac.uk/sigphon/

PREFACE

This workshop is the fifth meeting of computational phonologists under the auspices of SIGPHON. Our goal is to bring together researchers working in different theoretical frameworks and to advance the understanding of the computational and formal issues of current approaches to phonology.

Finite-state methods have been used very successfully to model classical phonological descriptions based on rewrite rules. It was C. Douglas Johnson (1972) who apparently first realized that phonological derivations could be modeled as a finite-state transduction in which each rule represents a regular relation between the input and the output languages. This is also the fundamental insight of Koskenniemi's two-level morphology (1983).

Although the rhetoric of Optimality Theory suggests that OT is fundamentally different from old-fashioned "serial" approaches to phonological description, there are many similarities. Ellison (1994) was the first to provide a computational account of OT using finite-state automata. In later works, it has been widely assumed that GEN is a regular relation. Many of the constraints that have been proposed in the OT literature can be formalized as a simple finite-state automaton or as an input-output transducer as in the two-level model.

However desirable it might be from a computational point of view, it is by no means evident that OT can be reduced to yet another way of encoding regular relations, except by limiting the number of violations that need to be counted and by restricting the type of allowable constraints. The question of what will turn out to be the appropriate way to model OT computationally is very much open at this point. Three of the papers in this volume address this very issue.

One of the reasons why OT appears to go beyond the finite-state domain is that much of the "cutting-edge" OT work in phonology deals with phenomena such as reduplication that were not considered as central in the earlier times. In this domain the distinction between phonology and morphology becomes blurred. Two of the papers in this volume address this topic, from very different points of view.

We thank our dedicated reviewers who enabled us to bring together the papers for these proceedings. We hope that you enjoy the workshop!

The Program Committee

PROGRAMME

9:00 : Welcome

9:05 : *Finite-State Non-Concatenative Morphotactics* (invited talk)
Kenneth R. Beesley and Lauri Karttunen

10:05 : *Temiar Reduplication in One-Level Prosodic Morphology*
Markus Walther

10:35 : Coffee

11:00 : *Easy and Hard Constraint Ranking in OT : Algorithms and Complexity*
Jason Eisner

11:30 : *Multi-Syllable Phonotactic Modelling*
Anja Belz

12:00 : Discussion of phonology/morphology papers from main conference

12:30 : Lunch

14:00 : *Approximation and Exactness in Finite State Optimality Theory* (invited talk)
Dale Gerdemann and Gertjan van Noord

15:00 : *Taking Primitive Optimality Theory Beyond the Finite State*
Daniel M. Albro

15:30 : Coffee

16:00 : *How to Design a Great Workbench Tool for Working Phonologists?* (panel)
Lauri Karttunen (moderator), Dan Albro, Ken Beesley, Jason Eisner, Dale Gerdemann, Arvi Hurskainen

17:00 : General discussion, Final remarks

TABLE OF CONTENTS

Preface iii
Programme iv
Table of contents v
Organisers and Programme Committee v
Short abstracts vi
Workshop Papers
Finite-State Non-Concatenative Morphotactics 1
Kenneth R. Beesley and Lauri Karttunen
Temiar Reduplication in One-Level Prosodic Morphology 13
Markus Walther
Easy and Hard Constraint Ranking in OT : Algorithms and Complexity 22
Jason Eisner
Approximation and Exactness in Finite State Optimality Theory.................................. 34
Dale Gerdemann and Gertjan van Noord
Multi-Syllable Phonotactic Modelling 46
Anja Belz
Taking Primitive Optimality Theory Beyond the Finite State 57
Daniel M. Albro

ORGANISERS AND PROGRAMME COMMITTEE

Lauri Karttunen (programme chair) Xerox Research Centre Europe
Markus Walther (local chair) University of Marburg
Jason Eisner (organization) University of Rochester
Alain Thériault (administration) Université de Montréal
Daniel Albro University of California at Los Angeles
Steven Bird University of Pennsylvania
John Coleman University of Oxford
Dan Jurafsky University of Colorado
András Kornai Belmont Research, Cambridge, MA

SHORT ABSTRACTS

Finite-State Non-Concatenative Morphotactics
Kenneth R. Beesley and Lauri Karttunen

A new finite-state technique, "compile-replace", lets a regular expression compiler reapply and modify its own output, freeing morphotactic description to use any finite-state operation. This provides an elegant solution for classic examples of non-concatenative phenomena in Malay and Arabic.

Temiar Reduplication in One-Level Prosodic Morphology
Markus Walther

This paper presents the first computational analysis of a difficult piece of prosodic morphology, aspectual reduplication in the Malaysian language Temiar, using the novel finite-state approach of One-Level Prosodic Morphology (Walther 1999b, 2000).

Easy and Hard Constraint Ranking in OT : Algorithms and Complexity
Jason Eisner

A simple version of the automatic constraint ranking problem is easier than previously known (linear on the number of constraints). But slightly more realistic versions are as bad as Σ_2-complete. Even checking a ranking against data is up to Δ_2-complete.

Approximation and Exactness in Finite State Optimality Theory
Dale Gerdemann and Gertjan van Noord

Frank & Satta (1998) showed that OT with gradient constraints generally is not finite-state. We present an improvement of the approximation of Karttunen (1998). The new method is exact and compact for the syllabification analysis of Prince and Smolensky (1993).

Multi-Syllable Phonotactic Modelling
Anja Belz

An approach to describing word-level phonotactics in terms of syllable classes. Such "multi-syllable" phonotactic models can be expressed in a formalism that facilitates automatic model construction and generalisation.

Taking Primitive Optimality Theory Beyond the Finite State
Daniel M. Albro

Extends the Primitive Optimality Theory formalism (Eisner 1997) to handle reduplication. Each candidate set becomes a Multiple Context-Free Language. Constraints, however, remain finite-state. Efficient candidate filtering is possible via an extended Earley's algorithm.

Panel Discussion: *How to Design a Great Workbench Tool for Working Phonologists?*
Moderator : Lauri Karttunen, co-author of the Xerox finite-state compiler

Dan Albro, author of the UCLA OTP tool
Kenneth R. Beesley, co-author of the Xerox finite-state compiler
Jason Eisner, author of the Primitive OT framework
Dale Gerdemann, co-author of the FSA Utilities toolbox
Arvi Hurskainen, author of tools for African languages

Finite-State Non-Concatenative Morphotactics

Kenneth R. Beesley and Lauri Karttunen
beesley@xrce.xerox.com, karttunen@xrce.xerox.com

Abstract

Finite-state morphology in the general tradition of the Two-Level and Xerox implementations has proved very successful in the production of robust morphological analyzer-generators, including many large-scale commercial systems. However, it has long been recognized that these implementations have serious limitations in handling non-concatenative phenomena. We describe a new technique for constructing finite-state transducers that involves reapplying the regular-expression compiler to its own output. Implemented in an algorithm called compile-replace, this technique has proved useful for handling non-concatenative phenomena; and we demonstrate it on Malay full-stem reduplication and Arabic stem interdigitation.

1 Introduction

Most natural languages construct words by concatenating morphemes together in strict orders. Such "concatenative morphotactics" can be impressively productive, especially in agglutinative languages like Aymara (Figure 1[1]) or Turkish, and in agglutinative/polysynthetic languages like Inuktitut (Figure 2)(Mallon, 1999, 2). In such languages a single word may contain as many morphemes as an average-length English sentence.

Finite-state morphology in the tradition of the Two-Level (Koskenniemi, 1983) and Xerox implementations (Karttunen, 1991; Karttunen, 1994; Beesley and Karttunen, 2000) has been very successful in implementing large-scale, robust and efficient morphological analyzer-generators for concatenative languages, including the commercially important European languages and non-Indo-European examples like Finnish, Turkish and Hungarian. However, Koskenniemi himself understood that his initial implementation had significant limitations in handling non-concatenative morphotactic processes:

> "Only restricted infixation and reduplication can be handled adequately with the present system. Some extensions or revisions will be necessary for an adequate description of languages possessing extensive infixation or reduplication" (Koskenniemi, 1983, 27).

This limitation has of course not escaped the notice of various reviewers, e.g. Sproat(1992). We shall argue that the morphotactic limitations of the traditional implementations are the direct result of relying solely on the concatenation operation in morphotactic description.

We describe a technique, within the Xerox implementation of finite-state morphology, that corrects the limitations at the source, going beyond concatenation to allow the full range of finite-state operations to be used in morphotactic description. Regular-expression descriptions are compiled into finite-state automata or transducers (collectively called networks) as usual, and then the compiler is re-applied to its own output, producing a modified but still finite-state network. This technique, implemented in an algorithm called COMPILE-REPLACE, has already proved useful for handling Malay full-stem reduplication and Arabic stem interdigitation, which will be described below. Before illustrating these applications, we will first outline our general approach to finite-state morphology.

[1] I wish to thank Stuart Newton for this example.

```
Lexical:  uta+ma+na-ka+p+xa+samacha-i+wa
Surface:  uta ma n  ka p xa samach  i wa

          uta      = house (root)
          +ma      = 2nd person possessive
          +na      = in
          -ka      = (locative, verbalizer)
          +p       = plural
          +xa      = perfect aspect
          +samacha = "apparently"
          -i       = 3rd person
          +wa      = topic marker
```

Figure 1: Aymara: *utamankapxasamachiwa* = "it appears that they are in your house"

```
Lexical:  Paris+mut+nngau+juma+niraq+lauq+sima+nngit+junga
Surface:  Pari  mu  nngau juma nira  lauq sima nngit tunga

          Paris     = (root = Paris)
          +mut      = terminalis case ending
          +nngau    = go (verbalizer)
          +juma     = want
          +niraq    = declare (that)
          +lauq     = past
          +sima     = (added to -lauq- indicates "distant past")
          +nngit    = negative
          +junga    = 1st person sing. present indic (nonspecific)
```

Figure 2: Inuktitut: *Parimunngaujumaniralauqsimanngittunga* = "I never said I wanted to go to Paris"

2 Finite-State Morphology

2.1 Analysis and Generation

In the most theory- and implementation-neutral form, morphological analysis and generation of written words can be modeled as a relation between the words themselves and analyses of those words. Computationally, as shown in Figure 3, a black-box module maps from words to analyses to effect Analysis, and from analyses to words to effect Generation.

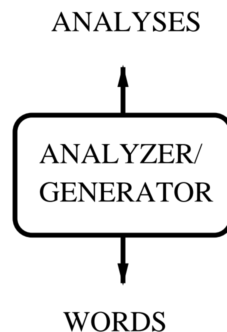


Figure 3: Morphological Analysis/Generation as a Relation between Analyses and Words

The basic claim or hope of the finite-state approach to natural-language morphology is that relations like that represented in Figure 3 are in fact regular relations, i.e. relations between two regular languages. The surface language consists of strings (= words = sequences of symbols) written according to some defined orthography. In a commercial application for a natural language, the surface language to be modeled is usually a given, e.g. the set of valid French words as written according to standard French orthography. The lexical language again consists of strings, but strings designed according to the needs and taste of the linguist, representing analyses of the surface words. It is sometimes convenient to design these lexical strings to show all the constituent morphemes in their morphophonemic form, separated and identified as in Figures 1 and 2. In other applications,

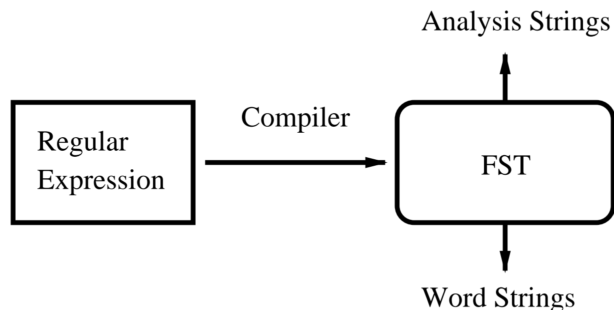


Figure 4: Compilation of a Regular Expression into an FST that Maps between Two Regular Languages

it may be useful to design the lexical strings to contain the traditional dictionary citation form, together with linguist-selected "tag" symbols like `+Noun`, `+Verb`, `+SG`, `+PL`, that convey category, person, number, tense, mood, case, etc. Thus the lexical string representing *paie*, the first-person singular, present indicative form of the French verb *payer* ("to pay"), might be spelled `payer+IndP+SG+P1+Verb`. The tag symbols are stored and manipulated just like alphabetic symbols, but they have multicharacter print names.

If the relation is finite-state, then it can be defined using the metalanguage of regular expressions; and, with a suitable compiler, the regular expression source code can be compiled into a finite-state transducer (FST), as shown in Figure 4, that implements the relation computationally. Following convention, we will often refer to the upper projection of the FST, representing analyses, as the LEXICAL language, a set of lexical strings; and we will refer to the lower projection as the SURFACE language, consisting of surface strings. There are compelling advantages to computing with such finite-state machines, including mathematical elegance, flexibility, and for most natural-language applications, high efficiency and data-compaction.

One computes with FSTs by applying them, in either direction, to an input string. When one such FST that was written for French is applied in an upward direction to the surface word *maisons* ("houses"), it returns the related string `maison+Fem+PL+Noun`, consisting of the citation form and tag symbols chosen by a linguist to convey that the surface form is a feminine noun in the plural form. A single surface string can be related to multiple lexical strings, e.g. applying this FST in an upward direction to surface string *suis* produces the four related lexical strings shown in Figure 5. Such ambiguity of surface strings is very common.

```
être+IndP+SG+P1+Verb
suivre+IndP+SG+P2+Verb
suivre+IndP+SG+P1+Verb
suivre+Imp+SG+P2+Verb
```

Figure 5: Multiple Analyses for *suis*

Conversely, the very same FST can be applied in a downward direction to a lexical string like `être+IndP+SG+P1+Verb` to return the related surface string *suis*; such transducers are inherently bidirectional. Ambiguity in the downward direction is also possible, as in the relation of the lexical string `payer+IndP+SG+P1+Verb` ("I pay") to the surface strings *paie* and *paye*, which are in fact valid alternate spellings in standard French orthography.

2.2 Morphotactics and Alternations

There are two challenges in modeling natural language morphology:

- Morphotactics
- Phonological/Orthographical Alternations

Finite-state morphology models both using regular expressions. The source descriptions may also be written in higher-level notations (e.g. **lexc** (Karttunen, 1993), **twolc** (Karttunen and Beesley, 1992) and Replace Rules (Karttunen, 1995; Karttunen, 1996; Kempe and Karttunen, 1996)) that are simply helpful shorthands for regular expressions and that compile, using their dedicated compilers, into finite-state

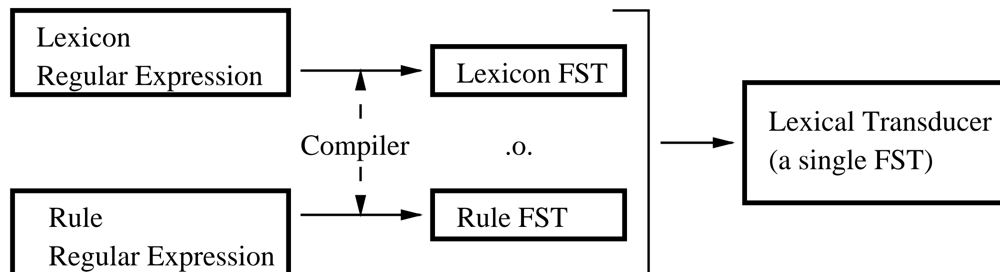


Figure 6: Creation of a Lexical Transducer

networks. In practice, the most commonly separated modules are a lexicon FST, containing lexical strings, and a separately written set of rule FSTs that map from the strings in the lexicon to properly spelled surface strings. The lexicon description defines the morphotactics of the language, and the rules define the alternations. The separately compiled lexicon and rule FSTs can subsequently be composed together as in Figure 6 to form a single "lexical transducer" (Karttunen et al., 1992) that could have been defined equivalently, but perhaps less perspicuously and less efficiently, with a single regular expression.

In the lexical transducers built at Xerox, the strings on the lower side of the transducer are inflected surface forms of the language. The strings on upper side of the transducer contain the citation forms of each morpheme and any number of tag symbols that indicate the inflections and derivations of the corresponding surface form. For example, the information that the comparative of the adjective *big* is *bigger* might be represented in the English lexical transducer by the path (= sequence of states and arcs) in Figure 7 where the zeros represent epsilon symbols.[2] The gemination of `g` and

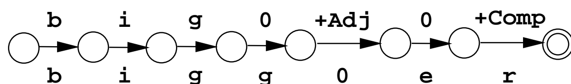


Figure 7: A Path in a Transducer for English

the epenthetical `e` in the surface form *bigger* result from the composition of the original lexicon FST with the rule FST representing the regular morphological alternations in English.

For the sake of clarity, Figure 7 represents the upper (= lexical) and the lower (= surface) side of the arc label separately on the opposite sides of the arc. In the remaining diagrams, we use a more compact notation: the upper and the lower symbol are combined into a single label of the form `upper:lower` if the symbols are distinct. A single symbol is used for an identity pair. In the standard notation, the path in Figure 7 is labeled as

`b i g 0:g +Adj:0 0:e +Comp:r`.

Lexical transducers are more efficient for analysis and generation than the classical two-level systems (Koskenniemi, 1983) because the morphotactics and the morphological alternations have been precompiled and need not be consulted at runtime. But it would be possible in principle, and perhaps advantageous for some purposes, to view the regular expressions defining the morphology of a language as an uncompiled "virtual network". All the finite-state operations (concatenation, union, intersection, composition, etc.) can be simulated by an apply routine at runtime.

Most languages build words by simply stringing morphemes (prefixes, roots and suffixes) together in strict orders. The morphotactic (word-building) processes of prefixation and suffixation can be straightforwardly modeled in finite state terms as concatenation. But some natural languages also exhibit non-concatenative morphotactics. Sometimes the languages themselves are called "non-concatenative languages", but most employ significant concatenation as well, so the term "not completely concatenative" (Lavie et al., 1988) is usually more appropriate.

[2]The epsilon symbols and their placement in the string are not significant. We will ignore them whenever it is convenient.

In Arabic, for example, prefixes and suffixes attach to stems in the usual concatenative way, but stems themselves are formed by a process known informally as interdigitation; while in Malay, noun plurals are formed by a process known as full-stem reduplication. Although Arabic and Malay also include prefixation and suffixation that are modeled straightforwardly by concatenation, a complete lexicon cannot be obtained without non-concatenative processes.

We will proceed with descriptions of how Malay reduplication and Semitic stem interdigitation are handled in finite-state morphology using the new compile-replace algorithm.

3 Compile-Replace

The central idea in our approach to the modeling of non-concatenative processes is to define networks using regular expressions, as before; but we now define the strings of an intermediate network so that they contain appropriate substrings that are themselves in the format of regular expressions. The compile-replace algorithm then reapplies the regular-expression compiler to its own output, compiling the regular-expression substrings in the intermediate network and replacing them with the result of the compilation.

To take a simple non-linguistic example, Figure 8 represents a network that maps the regular expression `a*` into `^[a*^]`; that is, the same expression enclosed between two special delimiters, `^[` and `^]`, that mark it as a regular-expression substring.

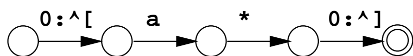


Figure 8: A Network with a Regular-Expression Substring on the Lower Side

The application of the compile-replace algorithm to the lower side of the network eliminates the markers, compiles the regular expression `a*` and maps the upper side of the path to the language resulting from the compilation. The network created by the operation is shown in Figure 9.

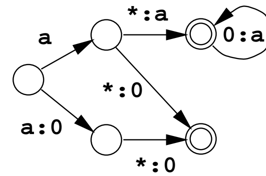


Figure 9: After the Application of Compile-Replace

When applied in the "upward" direction, the transducer in Figure 9 maps any string of the infinite `a*` language into the regular expression from which the language was compiled.

The compile-replace algorithm is essentially a variant of a simple recursive-descent copying routine. It expects to find delimited regular-expression substrings on a given side (upper or lower) of the network. Until an opening delimiter `^[` is encountered, the algorithm constructs a copy of the path it is following. If the network contains no regular-expression substrings, the result will be a copy of the original network. When a `^[` is encountered, the algorithm looks for a closing `^]` and extracts the path between the delimiters to be handled in a special way:

1. The symbols along the indicated side of the path are concatenated into a string and eliminated from the path leaving just the symbols on the opposite side.
2. A separate network is created that contains the modified path.
3. The extracted string is compiled into a second network with the standard regular-expression compiler.
4. The two networks are combined into a single one using the crossproduct operation.
5. The result is spliced between the states representing the origin and the destination of the regular-expression path.

After the special treatment of the regular-expression path is finished, normal processing is resumed in the destination state of the closing `^]` arc. For example, the result shown in Figure 9 represents the crossproduct of the two networks shown in Figure 10.

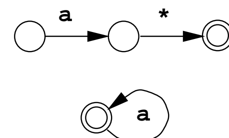


Figure 10: Networks Illustrating Steps 2 and 3 of the Compile-Replace Algorithm

```
Lexical:       b a g i   +Noun +Plural
Surface: ^[ { b a g i }          ^ 2   ^]

Lexical:       p e l a b u h a n   +Noun +Plural
Surface: ^[ { p e l a b u h a n }          ^ 2   ^]
```

Figure 11: Two Paths in the Initial Malay Transducer Defined via Concatenation

In this simple example, the upper language of the original network in Figure 8 is identical to the regular expression that is compiled and replaced. In the linguistic applications presented in the next sections, the two sides of a regular-expression path contain different strings. The upper side contains morphological information; the regular-expression operators appear only on the lower side and are not present in the final result.

3.1 Reduplication

Traditional Two-Level implementations are already capable of describing some limited reduplication and infixation as in Tagalog (Antworth, 1990, 156–162). The more challenging phenomenon is variable-length reduplication, as found in Malay and the closely related Indonesian language.

An example of variable-length full-stem reduplication occurs with the Malay stem *bagi*, which means "bag" or "suitcase"; this form is in fact number-neutral and can translate as the plural. Its overt plural is phonologically *bagibagi*,[3] formed by repeating the stem twice in a row. Although this pluralization process may appear concatenative, it does not involve concatenating a predictable pluralizing morpheme, but rather copying the preceding stem, whatever it may be and however long it may be. Thus the overt plural of *pelabuhan* ("port"), itself a derived form, is phonologically *pelabuhanpelabuhan.*

Productive reduplication cannot be described by finite-state or even context-free formalisms. It is well known that the copy language, $\{ww \mid w \in L\}$, where each word contains two copies of the same string, is a context-sensitive language. However, if the "base" language L is finite, we can construct a finite-state network that encodes L and the reduplications of all the strings in L. On the assumption that there are only a finite number of words subject to reduplication (no free compounding), it is possible to construct a lexical transducer for languages such as Malay. We will show a simple and elegant way to do this with strictly finite-state operations.

To understand the general solution to full-stem reduplication using the compile-replace algorithm requires a bit of background. In the regular expression calculus there are several operators that involve concatenation. For example, if `A` is a regular expression denoting a language or a relation, `A*` denotes zero or more and `A+` denotes one or more concatenations of `A` with itself. There are also operators that express a fixed number of concatenations. In the Xerox calculus, expressions of the form `A^n`, where `n` is an integer, denote n concatenations of `A`. `{abc}` denotes the concatenation of symbols `a`, `b`, and `c`. We also employ `^[` and `^]` as delimiter symbols around regular-expression substrings.

The reduplication of any string w can then be notated as `{w}^2`, and we start by defining a network where the lower-side strings are built by simple concatenation of a prefix `^[`, a root enclosed in braces, and an overt-plural suffix `^2` followed by the closing `^]`. Figure 11 shows the paths for two Malay plurals in the initial network.

The compile-replace algorithm, applied to the lower-side of this network, recognizes each individual delimited regular-expression substring like `^[{bagi}^2^]`, compiles it, and replaces it with the result of the compilation, here *bagibagi.* The same process applies to the entire lower-side language, resulting in a network that relates pairs of strings such as the ones in Figure 12. This provides the desired solution, still finite-state, for analyzing and generating full-stem reduplication in Malay.[4]

[3]In the standard orthography, such reduplicated words are written with a hyphen, e.g. *bagi-bagi*, that we will ignore for this example.

[4]It is well-known (McCarthy and Prince, 1995) that reduplication can be a more complex phenomenon than

```
Lexical: b a g i        +Noun +Plural
Surface: b a g i b a g i

Lexical: p e l a b u h a n                    +Noun +Plural
Surface: p e l a b u h a n p e l a b u h a n
```

Figure 12: The Malay FST After the Application of Compile-Replace to the Lower-Side Language

The special delimiters `^[` and `^]` can be used to surround any appropriate regular-expression substring, using any necessary regular-expression operators, and compile-replace may be applied to the lower-side and/or upper-side of the network as desired. There is nothing to stop the linguist from inserting delimiters multiple times, including via composition, and reapplying compile-replace multiple times (see the Appendix). The technique implemented in compile-replace is a general way of allowing the regular-expression compiler to reapply to and modify its own output.

3.2 Semitic Stem Interdigitation

3.2.1 Review of Earlier Work

Much of the work in non-concatenative finite-state morphotactics has been dedicated to handling Semitic stem interdigitation. An example of interdigitation occurs with the Arabic stem `katab`, which means "wrote". According to an influential autosegmental analysis (McCarthy, 1981), this stem consists of an all-consonant root `ktb` whose general meaning has to do with writing, an abstract consonant-vowel template `CVCVC`, and a voweling or vocalization that he symbolized simply as `a`, signifying perfect aspect and active voice. The root consonants are associated with the `C` slots of the template and the vowel or vowels with the `V` slots, producing a complete stem *katab*. If the root and the vocalization are thought of as morphemes, neither morpheme occurs continuously in the stem. The same root `ktb` can combine with the template `CVCVC` and a different vocalization `ui`, signifying perfect aspect and passive voice, producing the stem *kutib*, which means "was written". Similarly, the root `ktb` can combine with template `CVVCVC` and `ui` to produce *kuutib*, the root `drs` can combine with `CVCVC` and `ui` to form *duris*, and so forth.

Kay (1987) reformalized the autosegmental tiers of McCarthy (1981) as projections of a multi-level transducer and wrote a small Prolog-based prototype that handled the interdigitation of roots, CV-templates and vocalizations into abstract Arabic stems; this general approach, with multi-tape transducers, has been explored and extended by Kiraz in several papers (1994a; 1996; 1994b; 2000) with respect to Syriac and Arabic. The implementation is described in Kiraz and Grimley-Evans (1999).

In work more directly related to the current solution, it was Kataja and Koskenniemi (1988) who first demonstrated that Semitic (Akkadian) roots and patterns[5] could be formalized as regular languages, and that the non-concatenative interdigitation of stems could be elegantly formalized as the intersection of those regular languages. Thus Akkadian words were formalized as consisting of morphemes, some of which were combined together by intersection and others of which were combined via concatenation.

This was the key insight: morphotactic description could employ various finite-state operations, not just concatenation; and languages that required only concatenation were just special cases. By extension, the widely noticed limitations of early finite-state implementations in dealing with non-concatenative morphotactics could be traced to their dependence on the concatenation operation in morphotactic descriptions.

This insight of Kataja and Koskenniemi was applied by Beesley in a large-scale morphological analyzer for Arabic, first using an implementation that simulated the intersection of stems in code at runtime (Beesley, 1989; Beesley et al., 1989; Beesley, 1990; Beesley, 1991), and ran

it is in Malay. In some languages only a part of the stem is reduplicated and there may be systematic differences between the reduplicate and the base form. We believe that our approach to reduplication can account for these complex phenomena as well but we cannot discuss the issue here due to lack of space.

[5] These patterns combine what McCarthy (1981) would call templates and vocalizations.

rather slowly; and later, using Xerox finite-state technology (Beesley, 1996; Beesley, 1998a), a new implementation that intersected the stems at compile time and performed well at runtime. The 1996 algorithm that intersected roots and patterns into stems, and substituted the original roots and patterns on just the lower side with the intersected stem, was admittedly rather ad hoc and computationally intensive, taking over two hours to handle about 90,000 stems on a SUN Ultra workstation. The compile-replace algorithm is a vast improvement in both generality and efficiency, producing the same result in a few minutes.

Following the lines of Kataja and Koskenniemi (1988), we could define intermediate networks with regular-expression substrings that indicate the intersection of suitably encoded roots, templates, and vocalizations (for a formal description of what such regular-expression substrings would look like, see Beesley (1998c; 1998b)). However, the general-purpose intersection algorithm would be expensive in any non-trivial application, and the interdigitation of stems represents a special case of intersection that we achieve in practice by a much more efficient finite-state algorithm called MERGE.

3.2.2 Merge

The merge algorithm is a pattern-filling operation that combines two regular languages, a template and a filler, into a single one. The strings of the filler language consist of ordinary symbols such as `d`, `r`, `s`, `u`, `i`. The template expressions may contain special class symbols such as `C` (= consonant) or `V` (= vowel) that represent a predefined set of ordinary symbols. The objective of the merge operation is to align the template strings with the filler strings and to instantiate the class symbols of the template as the matching filler symbols.

Like intersection, the merge algorithm operates by following two paths, one in the template network, the other in the filler network, and it constructs the corresponding single path in the result network. Every state in the result corresponds to two original states, one in template, the other in the filler. If the original states are both final, the resulting state is also final; otherwise it is non-final. In other words, in order to construct a successful path, the algorithm must reach a final state in both of the original networks. If the new path terminates in a non-final state, it represents a failure and will eventually be pruned out.

The operation starts in the initial state of the original networks. At each point, the algorithm tries to find all the successful matches between the template arcs and filler arcs. A match is successful if the filler arc symbol is included in the class designated by the template arc symbol. The main difference between merge and classical intersection is in Conditions 1 and 2 below:

1. If a successful match is found, a new arc is added to the current result state. The arc is labeled with the filler arc symbol; its destination is the result state that corresponds to the two original destinations.
2. If no successful match is found for a given template arc, the arc is copied into the current result state. Its destination is the result state that corresponds to the destination of the template arc and the current filler state.

In effect, Condition 2 preserves any template arc that does not find a match. In that case, the path in the template network advances to a new state while the path in the filler network stays at the current state.

We use the networks in Figure 13 to illustrate the effect of the merge algorithm. Figure 13 shows a linear template network and two filler networks, one of which is cyclic.

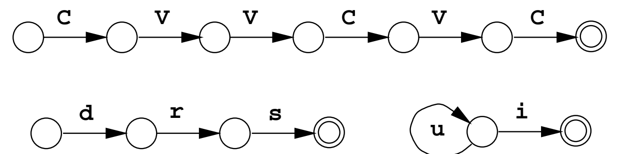


Figure 13: A Template Network and Two Filler Networks

It is easy to see that the merge of the `drs` network with the template network yields the result shown in Figure 14. The three symbols of the filler string are instantiated in the three consonant slots in the `CVVCVC` template.

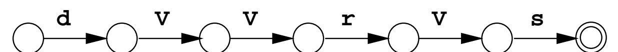


Figure 14: Intermediate Result.

```
Lexical:      k t b =Root C V C V C =Template a + =Voc
Surface:   ^[ k t b  .m>. C V C V C    .<m.   a +      ^]

Lexical:      k t b =Root C V C V C =Template u * i =Voc
Surface:   ^[ k t b  .m>. C V C V C    .<m.   u * i      ^]

Lexical:      d r s =Root C V V C V C =Template u * i =Voc
Surface:   ^[ d r s  .m>. C V V C V C   .<m.    u * i      ^]
```

Figure 16: Initial paths

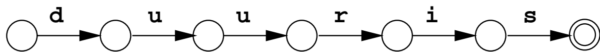


Figure 15: Final Result

Figure 15 presents the final result in which the second filler network in Figure 13 is merged with the intermediate result shown in Figure 14.

In this case, the filler language contains an infinite set of strings, but only one successful path can be constructed. Because the filler string ends with a single i, the first two V symbols can be instantiated only as u. Note that ordinary symbols in the partially filled template are treated like the class symbols that do not find a match. That is, they are copied into the result in their current position without consuming a filler symbol.

To introduce the merge operation into the Xerox regular expression calculus we need to choose an operator symbol. Because merge, like subtraction, is a non-commutative operation, we also must distinguish between the template and the filler. For example, we could choose .m. as the operator and decide by convention which of the two operands plays which role in expressions such as [A .m. B]. What we actually have done, perhaps without a sufficiently good motivation, is to introduce two variants of the merge operator, .<m. and .m>., that differ only with respect to whether the template is to the left (.<m.) or to the right (.m>.) of the the filler. The expression [A .<m. B] represents the same merge operation as [B .m>. A]. In both cases, A denotes the template, B denotes the filler, and the result is the same. With these new operators, the network in Figure 15 can be compiled from an expression such as

```
d r s .m>. C V V C V C .<m. u* i
```

As we have defined them, .<m. and .m>. are weakly binding left-associative operators. In this example, the first merge instantiates the filler consonants, the second operation fills the vowel slots. However, the order in which the merge operations are performed is irrelevant in this case because the two filler languages do not provide competing instantiations for the same class symbols.

3.2.3 Merging Roots and Vocalizations with Templates

Following the tradition, we can represent the lexical forms of Arabic stems as consisting of three components, a consonantal root, a CV template and a vocalization, possibly preceded and followed by additional affixes. In contrast to McCarthy, Kay, and Kiraz, we combine the three components into a single projection. In a sense, McCarthy's three tiers are conflated into a single one with three distinct parts. In our opinion, there is no substantive difference from a computational point of view.

For example, the initial lexical representation of the surface forms *katab*, *kutib*, and *duuris*, may be represented as a concatenation of the three components shown in Figure 16. We use the symbols =Root, =Template, and =Voc to designate the three components of the lexical form. The corresponding initial surface form is a regular-expression substring, containing two merge operators, that will be compiled and replaced by the interdigitated surface form.

The application of the compile-replace operation to the lower side of the initial lexicon yields a transducer that maps the Arabic interdigitated forms directly into their corresponding tripartite analyses and vice versa, as illustrated in Figure 17.

Alternation rules are subsequently composed on the lower side of the result to map the interdigitated, but still morphophonemic, strings into real surface strings.

```
Lexical:  k t b =Root C V C V C =Template a + =Voc
Surface:              k a t a b

Lexical:  k t b =Root C V C V C =Template u * i =Voc
Surface:              k u t i b

Lexical:  d r s =Root C V V C V C =Template u * i =Voc
Surface:              d u u r i s
```

Figure 17: After Applying Compile-Replace to the Lower Side

Although many Arabic templates are widely considered to be pure CV-patterns, it has been argued that certain templates also contain "hard-wired" specific vowels and consonants.[6] For example, the so-called "FormVIII" template is considered, by some linguists, to contain an embedded `t`: `CtVCVC`.

The presence of ordinary symbols in the template does not pose any problem for the analysis adopted here. As we already mentioned in discussing the intermediate representation in Figure 14, the merge operation treats ordinary symbols in a partially filled template in the same manner as it treats unmatched class symbols. The merge of a root such as `ktb` with the presumed FormVIII template and the `a+` vocalism,

```
k t b .m>. C t V C V C .<m. a+
```

produces the desired result, `ktatab`, without any additional mechanism.

4 Status of the Implementations

4.1 Malay Morphological Analyzer/Generator

Malay and Indonesian are closely-related languages characterized by rich derivation and little or nothing that could be called inflection. The Malay morphological analyzer prototype, written using **lexc**, Replace Rules, and compile-replace, implements approximately 50 different derivational processes, including prefixation, suffixation, prefix-suffix pairs (circumfixation), reduplication, some infixation, and combinations of these processes. Each root is marked manually in the source dictionary to indicate the idiosyncratic subset of derivational processes that it undergoes.

The small prototype dictionary, stored in an XML format, contains approximately 1000 roots, with about 1500 derivational subentries (i.e. an average of 1.5 derivational processes per root). At compile time, the XML dictionary is parsed and "downtranslated" into the source format required for the **lexc** compiler. The XML dictionary could be expanded by any competent Malay lexicographer.

4.2 Arabic Morphological Analyzer/Generator

The current Arabic system has been described in some detail in previous publications (Beesley, 1996; Beesley, 1998a; Beesley, 1998b) and is available for testing on the Internet.[7] The modification of the system to use the compile-replace algorithm has not changed the size or the behavior of the system in any way, but it has reduced the compilation time from hours to minutes.

5 Conclusion

The well-founded criticism of traditional implementations of finite-state morphology, that they are limited to handling concatenative morphotactics, is a direct result of their dependence on the concatenation operation in morphotactic description. The technique described here, implemented in the compile-replace algorithm, allows the regular-expression compiler to reapply to and modify its own output, effectively freeing morphotactic description to use any finite-state operation. Significant experiments with Malay and a much larger application in Arabic have shown the value of this technique in handling two classic examples of non-concatenative morphotactics: full-stem reduplication and Semitic stem interdigitation. Work remains to be done in applying the technique to other known varieties of non-concatenative morphotactics.

[6] See Beesley (1998c) for a discussion of this controversial issue.

[7] http://www.xrce.xerox.com/research/mltt/arabic/

The compile-replace algorithm and the merge operator introduced in this paper are general techniques not limited to handling the specific morphotactic problems we have discussed. We expect that they will have many other useful applications. One illustration is given in the Appendix.

6 Appendix: Palindrome Extraction

To demonstrate the power of the compile-replace method, let us show how it can be applied to solve another "hard" problem: identifying and extracting all the palindromes from a lexicon. Like reduplication, palindrome identification appears at first to require more powerful tools than a finite-state calculus. But this task can be accomplished, in fact quite efficiently, by using the compile-replace technique.

Let us assume that `L` is a simple network constructed from an English wordlist. We start by extracting from `L` all the words with a property that is necessary but not sufficient for being a palindrome, namely, the words whose inverse is also an English word. This step can be accomplished by redefining `L` as `[L & L.r]` where `&` represents intersection and `.r` is the reverse operator. The resulting network contains palindromes such as `madam` as well non-palindromes such as `dog` and `god`.

The remaining task is to eliminate all the words like `dog` that are not identical to their own inverse. This can be done in three steps. We first apply the technique used for Malay reduplication. That is, we redefine `L` as `"^[" "[" L XX "]" "^" 2 "^]"`, and apply the compile-replace operation. At this point the lower-side of L contains strings such as `dogXXdogXX` and `madamXXmadamXX` where `XX` is a specially introduced symbol to mark the middle (and the end) of each string.

The next, and somewhat delicate, step is to replace the `XX` markers by the desired operators, intersection and reverse, and to wrap the special regular expression delimiters `^[` and `^]` around the whole lexicon. This can be done by composing `L` with one or several replace transducers to yield a network consisting of expressions such as `^[ d o g & [d o g].r ^]` and `^[ m a d a m & [m a d a m].r ^]`

In the third and final step, the application of compile-replace eliminates words like `dog` because the intersection of `dog` with the inverted form `god` is empty. Only the palindromes survive the operation. The extraction of all the palindromes from the 25K Unix /usr/dict/words file by this method takes a couple of seconds.

References

Evan L. Antworth. 1990. *PC-KIMMO: a two-level processor for morphological analysis.* Number 16 in Occasional publications in academic computing. Summer Institute of Linguistics, Dallas.

Kenneth R. Beesley and Lauri Karttunen. 2000. *Finite-State Morphology: Xerox Tools and Techniques.* Cambridge University Press. Forthcoming.

Kenneth R. Beesley, Tim Buckwalter, and Stuart N. Newton. 1989. Two-level finite-state analysis of Arabic morphology. In *Proceedings of the Seminar on Bilingual Computing in Arabic and English*, Cambridge, England, September 6-7. No pagination.

Kenneth R. Beesley. 1989. Computer analysis of Arabic morphology: A two-level approach with detours. In *Third Annual Symposium on Arabic Linguistics*, Salt Lake City, March 3-4. University of Utah. Published as Beesley, 1991.

Kenneth R. Beesley. 1990. Finite-state description of Arabic morphology. In *Proceedings of the Second Cambridge Conference on Bilingual Computing in Arabic and English*, September 5-7. No pagination.

Kenneth R. Beesley. 1991. Computer analysis of Arabic morphology: A two-level approach with detours. In Bernard Comrie and Mushira Eid, editors, *Perspectives on Arabic Linguistics III: Papers from the Third Annual Symposium on Arabic Linguistics*, pages 155–172. John Benjamins, Amsterdam. Read originally at the Third Annual Symposium on Arabic Linguistics, University of Utah, Salt Lake City, Utah, 3-4 March 1989.

Kenneth R. Beesley. 1996. Arabic finite-state morphological analysis and generation. In *COLING'96*, volume 1, pages 89–94, Copenhagen, August 5-9. Center for Sprogteknologi. The 16th International Conference on Computational Linguistics.

Kenneth R. Beesley. 1998a. Arabic morphologi-

cal analysis on the Internet. In *ICEMCO–98*, Cambridge, April 17-18. Centre for Middle Eastern Studies. Proceedings of the 6th International Conference and Exhibition on Multilingual Computing. Paper number 3.1.1; no pagination.

Kenneth R. Beesley. 1998b. Arabic morphology using only finite-state operations. In Michael Rosner, editor, *Computational Approaches to Semitic Languages: Proceedings of the Workshop*, pages 50–57, Montréal, Québec, August 16. Université de Montréal.

Kenneth R. Beesley. 1998c. Arabic stem morphotactics via finite-state intersection. Paper presented at the 12th Symposium on Arabic Linguistics, Arabic Linguistic Society, 6-7 March, 1998, Champaign, IL.

Lauri Karttunen and Kenneth R. Beesley. 1992. Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA, October.

Lauri Karttunen, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-level morphology with composition. In *COLING'92*, pages 141–148, Nantes, France, August 23-28.

Lauri Karttunen. 1991. Finite-state constraints. In *Proceedings of the International Conference on Current Issues in Computational Linguistics*, Penang, Malaysia, June 10-14. Universiti Sains Malaysia.

Lauri Karttunen. 1993. Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center, Palo Alto, CA, April.

Lauri Karttunen. 1994. Constructing lexical transducers. In *COLING'94*, Kyoto, Japan.

Lauri Karttunen. 1995. The replace operator. In *ACL'95*, Cambridge, MA. cmp-lg/9504032.

Lauri Karttunen. 1996. Directed replacement. In *ACL'96*, Santa Cruz, CA. cmp-lg/9606029.

Laura Kataja and Kimmo Koskenniemi. 1988. Finite-state description of Semitic morphology: A case study of Ancient Akkadian. In *COLING'88*, pages 313–315.

Martin Kay. 1987. Nonconcatenative finite-state morphology. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 2–10.

André Kempe and Lauri Karttunen. 1996. Parallel replacement in finite-state calculus. In *COLING'96*, Copenhagen, August 5–9. cmp-lg/9607007.

George Anton Kiraz and Edmund Grimley-Evans. 1999. Multi-tape automata for speech and language systems: A Prolog implementation. In Jean-Marc Champarnaud, Denis Maurel, and Djelloul Ziadi, editors, *Automata Implementation*, volume 1660 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, Germany.

George Anton Kiraz. 1994a. Multi-tape two-level morphology: a case study in Semitic non-linear morphology. In *COLING'94*, volume 1, pages 180–186.

George Anton Kiraz. 1994b. Multi-tape two-level morphology: a case study in Semitic non-linear morphology. In *Proceedings of the 15th International Conference on Computational Linguistics*, Kyoto, Japan.

George Anton Kiraz. 1996. Semhe: A generalised two-level system. In *Proceedings of the 34th Annual Meeting of the Association of Computational Linguistics*, Santa Cruz, CA.

George Anton Kiraz. 2000. Multi-tiered non-linear morphology: A case study on Semitic. *Computational Linguistics*, 26(1).

Kimmo Koskenniemi. 1983. Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

Alon Lavie, Alon Itai, Uzzi Ornan, and Mori Rimon. 1988. On the applicability of two level morphology to the inflection of Hebrew verbs. In *Proceedings of ALLC III*, pages 246–260.

Mick Mallon. 1999. Inuktitut linguistics for technocrats. Technical report, Ittukuluuk Language Programs, Iqaluit, Nunavut, Canada.

John J. McCarthy and Alan Prince. 1995. Faithfulness and reduplicative identity. Occasional papers in Linguistics 18, University of Massachusetts, Amherst, MA. ROA-60.

John J. McCarthy. 1981. A prosodic theory of nonconcatenative morphology. *Linguistic Inquiry*, 12(3):373–418.

Richard Sproat. 1992. *Morphology and Computation.* MIT Press, Cambridge, MA.

Temiar Reduplication in One-Level Prosodic Morphology

Markus Walther
University of Marburg
FB09/IGS, Wilhelm-Röpke-Str. 6A, D-35032 Marburg, Germany
Markus.Walther@mailer.uni-marburg.de

Abstract

Temiar reduplication is a difficult piece of prosodic morphology. This paper presents the first computational analysis of Temiar reduplication, using the novel finite-state approach of One-Level Prosodic Morphology originally developed by Walther (1999b, 2000). After reviewing both the data and the basic tenets of One-level Prosodic Morphology, the analysis is laid out in some detail, using the notation of the FSA Utilities finite-state toolkit (van Noord 1997). One important discovery is that in this approach one can easily define a regular expression operator which ambiguously scans a string in the left- or rightward direction for a certain prosodic property. This yields an elegant account of base-length-dependent triggering of reduplication as found in Temiar.

1 Introduction

Temiar is an Austroasiatic language of the Mon-Khmer group spoken by a variety of tribal people in West Malaysia (Benjamin, 1976). Its intricate morphological system has received some attention in the theoretical literature. The main focus has been on the aspectual morphology of verbs, where an interesting pattern of partial reduplication emerges that is sensitive to the size of the verbal root. For example, in the active continuative, *gɛlgəl* 'to eat' reduplicates both the initial /g/ and the final /l/ of its monosyllabic base *gəl*. In contrast, bisyllabic *səluh* 'to shoot' comes out as *sɛhluh*, where only the final /h/ is copied, this time as an infix.

Temiar reduplication thus appears to be a suitably rich testing ground for a novel approach to reduplication developed by (Walther, 1999b; Walther, 2000) within a finite-state framework. Even though that approach, One-Level Prosodic Morphology, was presented from the outset as being generally applicable, it has been proven time and time again that only concrete empirical application of a particular approach to computational morphology and phonology will fully reveal its inherent virtues and weaknesses. As an example, (Beesley, 1998) reports that it was actual experimentation with grammars of word-formation in Arabic and Hungarian which fully revealed the negative effects of modelling long-distance circumfixional dependencies in purely finite-state terms, subsequently leading to some suggestions for improvement.

It is perhaps worth emphasizing that (Walther, 1999b)'s solution for reduplication in a finite-state context is preferrable for cross-linguistic validation precisely because it is the first that solves the problem in the *general* case. Because reduplication often involves copying of a strictly bounded amount of material, the bounded case *could* in principle be modelled as a finite-state process by enumerating all possible forms of the copy and then making sure each was matched to the proper stem. To solve this simplified problem, no new techniques are needed in theory. In practice however, the brute-force enumeration approach apparently has not been pursued further, apart from isolated examples (see Antworth (1990), p.157f for a fixed-size case in Tagalog). This is probably because such an approach is awkward to specify in actual grammars and because it will inevitably lead to an explosion of the state space (Sproat (1992), p.161). Finally and in contrast to (Walther, 1999b), it would clearly break down for *productive* total reduplication, which is isomorphic to the context-sensitive language $\{ww|w \in \Sigma^+\}$.

A second motivation for choosing Temiar is that all prior analyses of its data are heavily underformalized and incomplete, irrespective of whether they are situated in the older rule paradigm (McCarthy, 1982; Broselow and McCarthy, 1983; Sloan, 1988; Shaw, 1993) or an optimality-theoretic setting (Gafos, 1995; Gafos, 1996; Gafos, 1998b; Gafos, 1998a). Hence a formalized and computationally tested analysis that strives to keep a healthy balance

with respect to linguistic adequacy would represent significant progress on its own.

In the rest of the paper I will attempt to provide just such an analysis, beginning in §2 with a presentation of the relevant data. Next, section §3 reviews the core of One-Level Prosodic Morphology, which will be used as formal background. Using that background, the analysis is then fully developed in §4. The paper concludes with some discussion in §5.

2 Temiar reduplication

All data on Temiar reduplication in this section come from (Benjamin, 1976), the main source on the subject.[1] According to Benjamin, the characteristic aspectual paradigms of "monosyllabic and schewa-form verbs" (B:168) are as follows (B:169):

(1)

	'to call'	'to lie down/sleep/marry'	
	'monosyllabic'	'schewa-form'	
active	ˈkɔɔw	sə.ˈlɔg	perfective
	ka.ˈkɔɔw	sa.ˈlɔg	simulfactive
	kɛw.ˈkɔɔw	sɛ**g**.ˈlɔg	continuative
caus.	tɛr.ˈkɔɔw	sɛr.ˈlɔg	perfective
	tə.ra.ˈkɔɔw	sə.ra.ˈlɔg	simulfactive
	tə.rɛ**w**.ˈkɔɔw	sə.rɛ**g**.ˈlɔg	continuative

We have inferred syllabifications in (1) from the statement that "only two types of syllables occur: *open syllables* of canonical form CV, and *closed syllables* of canonical form CVC" (B:141). Note that Benjamin abstracts from vowel length here. Word-level stress, which is "falling regularly on the final syllable" (B:139), is likewise inferred in (1). Observe that only monosyllabic roots like *kɔɔw* reduplicate their initial consonant in the non-perfective aspectual forms of the active, while longer roots like *səlɔg* do not. This contrasts with obligatory reduplication of the root-final consonant in the continuative.

An important further generalization is that all extra segmental material beyond the bare root is inserted immediately before the stressed syllable, leading to prefixation for monosyllabic roots, but infixation in polysyllabic ones (Gafos, 1998b). From this point of view we can also see a correlation between the fact that causative forms of monosyllabic roots – which must be at least bisyllabic – begin with a fixed /t/[2] and the restriction that words must "always begin and end with a consonant" (B:141). In triconsonantal roots like *səlɔg* that restriction is taken care of by the first root consonant itself, so no fixed segment needs to appear.

According to Benjamin, prefinal syllables – which are unstressed – can show alternation of their vocalic quality: "In prefinal closed syllables the inner vowels /e ə o/ are replaced by the outer vowels /i ɛ u/ respectively" (B:144). This descriptive generalization accounts for the remaining contrasts in (1), witness e.g. *sə.lɔg* versus *sɛg.lɔg*.

It is interesting to see that Temiar even exhibits phonological modifications between base and reduplicant, affecting consonants in the continuative:

(2)	yaap	→	**yɛm**.yaap	'to cry' (B:143)
	pət	→	**pɛn**.pət	'to long for' (B:146)
	sə.lɔɔk	→	sɛ**ŋ**.lɔɔk	'to hunt successfully' (B:146)

Benjamin explains that medial coda consonants from the class of oral voiceless stops turn into their voiced nasal equivalents in Northern Temiar (and to plain voiced stops in the Southern dialect; B:143).

It is of some importance to clarify a number of further aspects of the data and their interpretation. First, theorists have frequently employed the stronger term 'minor syllables' for Benjamin's prefinal syllables, reflecting their alleged special status by means of an impoverished representation (e.g. empty syllable nuclei in (Gafos, 1998b)) and/or further formal mechanisms (e.g. a ban on full vowels in prefinal position *PREFINAL-V (Gafos, 1998a)). We do not follow this move here, because empirically it is neither true that penultimate vowels are categorically restricted to schwa-like vowels (*halab* 'to go downriver', *sindul* 'to float', etc.) nor are there any solid statistics of a presumed tendency to vowel reduction in unstressed syllables, nor can the variable quality of prefinal vowels be consistently derived from flanking consonants. Hence, such penultimative vowels are to be lexically specified as alternating.

Second, Benjamin's subclass restriction of (1) to "monosyllabic and schewa-form verbs" correctly excludes polysyllabic roots like the already mentioned *halab* and *sindul*, where prefinal open syllables with vowels outside of /e ə o/ occur. These roots undergo "very few morphological changes" (B:170), basically proclitization.

[1] We will abbreviate further references to this work with "(B: <page number>)" in the text. Moreover, to highlight reduplicated parts in the data they will often be printed in bold.

[2] Or /b/, if the root starts in /c,t/: /caaʔ/ 'to eat' gives /bɛr.caaʔ/ 'to feed' (B:169).

Third, paradigms for a given root are hardly ever complete, with various irregularities and non-productive patterns also occuring (B:169f). Again, a good deal of lexicalization would seem necessary to correctly describe Temiar verbs in a realistic grammar fragment.

Given this descriptive summary, our goals for the upcoming analysis are, first, to treat the *full* paradigm of (1). As a second goal, we would like to reflect the emergent formal desiderata in a transparent way, in particular referring to the need to account for *repetition, truncation, infixation* and *phonological modification*. Thirdly, we will attempt a *compositional* analysis of the morphological exponency of aspect.

3 One-Level Prosodic Morphology

In order to provide the necessary background for the Temiar analysis in §4, this section briefly reviews the finite-state approach to prosodic morphology developed in (Walther, 1999b),

That work itself was presented as an extension to (Bird and Ellison, 1994)'s One-Level Phonology framework, where phonological representations, morphemes and more abstract generalizations are all finite-state automata that express surface-true constraints on word forms, and constraint combination is by automata intersection.

In a nutshell, the extension comprises three main components. We (i) represent phonological strings differently for purposes of modelling prosodic morphology, (ii) implement reduplicative coyping by automata intersection, and (iii) introduce a resource-conscious variant of automata.

For (i), operators are provided that construct enriched automata from a simple string automaton, in particular giving it a kind of doubly-linked structure so that the symbol repetition inherent in reduplication translates into following backwards-pointing technical transitions. The individual enrichments involve only local computation per state or transition, so that on-the-fly implementation is easy if desired. In other words, one does not necessarily have to enrich the entire lexicon in advance.

Enriched representations In a bit more detail, the enrichments of (i) are as follows. The three aspects of *reduplication* or symbol repetition, *truncation* or symbol skipping and *infixation* or transitive, non-immediate precedence of symbols are reflected in three regular expression operators, $add_repeats, add_skips, add_self_loops$. Each takes the underlying automaton A of a regular language L_A as its only argument. Formally, they can be defined as follows:

(3) Let $A = (Q, \Sigma, \delta, q_0, F)$ be the minimal ϵ-free[3] finite-state automaton for L_A, with Q a finite set of states, finite alphabet Σ, transition function $\delta : Q \times \Sigma \mapsto 2^Q$, start state $q_0 \in Q$ and set of final states $F \subseteq Q$.

a. Assume $repeat \notin \Sigma$.
$add_repeats(A) \stackrel{def}{=} (Q, \Sigma', \delta', q_0, F)$,
where $\Sigma' = \Sigma \cup \{repeat\}$,
$\forall x \in \Sigma\, \forall q \in Q$: $\delta'(q, x) = \delta(q, x)$ and
$\forall p \in Q$:$\delta'(p, repeat) = \{q | \, p \in \delta(q, x)\}$

b. Assume $skip \notin \Sigma$.
$add_skips(A) \stackrel{def}{=} (Q, \Sigma', \delta', q_0, F)$,
where $\Sigma' = \Sigma \cup \{skip\}$,
$\forall x \in \Sigma\, \forall q \in Q$: $\delta'(q, x) = \delta(q, x)$ and
$\forall q \in Q$: $\delta'(q, skip) = \delta(q, x)$

c. $add_self_loops(A) \stackrel{def}{=} (Q, \Sigma, \delta', q_0, F)$,
where
$\delta' = \delta \cup \{(q, \sigma, \{q\}) | \, q \in Q, \, \sigma \in \Sigma\}$

An example enrichment of Temiar *səlɔg* is shown in figure 1. One can imagine how $skip$ and $repeat$ transitions allow, figuratively speaking, forward and backward movement within a string, while self loops will absorb infixal morphemes that are intersected with fig. 1. Finally, so-called *synchronization bits* **:1, :0** were introduced in (Walther, 1999b) to define the extent of a reduplicative base constituent in a segment-independent way. Bit value **:1** marks the edges and **:0** the interior segments of a base, as shown in fig. 1 for a hypothetical whole-root reduplication pattern. In actual practive, synchronization

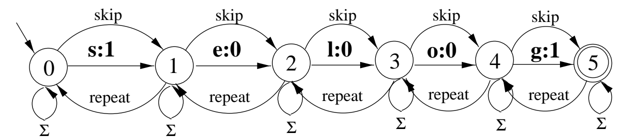


Figure 1: add_repeats(add_skips(add_self_loops(selog)))

bits are sets of symbols, just like the rest of the alphabet. Sets as transition labels improve over traditional automata in terms of automata compactness, were already proposed for phonology in (Bird

[3]Minimality prevents non-(co)-accessible transitions from getting enriched, while lack of ϵ transitions keeps positional $skip/repeat$ 'movement' in lockstep with segmental positions.

and Ellison, 1992) and do not increase mathematical expressivity beyond regular languages.[4] Hence, the segmental part of fig. 1 may be defined in a modular fashion through the intersection of strings of symbol sets that mention only certain dimensions (here: phonemes and synchronisation bits), being underspecified for the unmentioned dimensions. We will again follow (Walther, 1999b) in conceiving of sets as types arranged in a type hierarchy that is structured by set inclusion, and also in allowing arbitrary boolean combinations of types.

Copying as intersection Given enriched representations as in fig. 1, various patterns of reduplication are now easy to define. We can denote a synchronised abstract string by the regular expression

$$base \equiv seg{:}1\; seg{:}0^{*}\; seg{:}1$$

where seg is the type subsuming all phonological segments. Then hypothetical total reduplication – unattested in Temiar, but wellknown from Indonesian and many other languages – is described by

$$total \equiv base\, \boldsymbol{repeat}^{*}\, base$$

A variant slightly more akin to Temiar – and actually attested in the neighbouring language Semai – that $skip$s the interior of the base in a prefixed reduplicant is just as easy:

$$semai \equiv seg{:}1\; \boldsymbol{skip}^{*}\; seg{:}1\; \boldsymbol{repeat}^{*}\; base$$

Ignoring self loops for the moment, all we need now to apply a reduplication pattern to an enriched base representation is simply to *intersect* the former with the latter: automata intersection has sufficient formal power to implement reduplicative copying! Here is an example, using the abbreviation $selog \equiv \boldsymbol{s{:}1}\; \boldsymbol{e{:}0}\; \boldsymbol{l{:}0}\; \boldsymbol{o{:}0}\; \boldsymbol{g{:}1}$ for perspicuous display:

$$add_repeats(selog) \cap total \equiv selog\, \boldsymbol{repeat^{5}}\, selog$$

As pointed out in (Walther, 2000), generalizing to a *set* of bases involves nothing more than enriching each base separately, then forming the union of the resulting automata. The opposite order would produce unwanted cross-string repetition, since $add_repeats$ does not distribute over union. However, an unpublished experiment shows that on-demand implementation of a slightly modified $add_repeats$ can help to preserve the memory efficiency of building a minimized base lexicon as the union of individual base strings first. Due to lack of space, the details will be reported elsewhere.

[4] Of course, the identity requirement for matching transitions in traditional automata intersection must be replaced by a non-empty intersection requirement for set-based matching.

Resource consciousness As much as we need the formal means provided by self loops for infixations like Temiar *s-a-lɔg*, the resulting automata overgenerate massively. What's missing according to (Walther, 1999b) is a distinction between explicitly contributed, independent information (e.g. the infix *-a-* itself) and contextual, dependent information that is tolerated but must be provided by other constraints (e.g. the $1 \xrightarrow{\Sigma} 1$ self loop that *hosts* the infix). Therefore, a parallel distinction between two kinds of symbols – producers and consumers – was introduced. In that scenario a symbol represents an information resource that needs to be produced at least once, then can be consumed arbitrarily often. To utilize the distinction, an additional P/C bit accompanies symbols, with P/C = 1 for producers. All symbols introduced by the three enrichment operators are consumers. Furthermore, automata intersection is made aware of these resource-conscious notions by splitting it into two variants: In open interpretation mode, P/C bits of matching symbols are combined by logical OR, so that a result transition will be marked as a producer whenever at least one argument transition is a producer. In closed interpretation mode, combination is by logical AND instead, allowing only producer-producer matches. Grammatical evaluation can then be characterized as follows:

$$(\text{Lexicon} \cap_{open} \text{Constraint}_1 \cdots \cap_{open} \text{Constraint}_N) \cap_{closed} \boldsymbol{\Sigma}^{*}$$

Here and elsewhere, producers are in bold print. Note the final intersection with the universal producer language, which eliminates unused consumer transitions, the main source of overgeneration.

4 The analysis

We have assembled enough background now to proceed to the actual analysis of the Temiar data in (1). The analysis is implemented using FSA Utilities, a finite-state toolbox written in Prolog which encourages rapid prototyping (van Noord, 1997). Figure 2 shows a relevant fragment of its syntax (extensions and modifications in italics).

In displaying the grammar, we will take liberty in suppressing certain definitions in the interest of conciseness, relying on the mnemonic value of

`{}`	empty language
`[E1,E2, ... ,En]`	concatenation
`{E1,E2, ... ,En}`	union
`E*`	Kleene closure
`E^`	optionality
`E1 & E2`	intersection
`A -l-> / -r-> ( B / C)`	monotonic rules
`~S`	set complement
Head(arg1, ... , argN) := Body	macro def.

Figure 2: Regular expression operators

their names instead. A case in point is `producer(T)`, `consumer(T)`: since the names are self-explanatory, it suffices to note that the only argument `T` contains type formulae that denote the symbol sets, as explained before. Allowable type-combining operators are conjunction `&`, disjunction `;` and negation ~. The same goes for monotonic rules, which – unlike rewrite rules – can only specialize their focussed segmental position `A` to `B`. They exist in two variants, where `A -r-> B/C` notates the case where context `C` is right-adjacent to the focus ($A \rightarrow B/__C$), and vice versa for `A -l-> B/C`.

Syllabification To define the reduplicant in prosodic terms later on, we need `syllabification` in the first place. Here a simplified finite-state version of a proposal by (Walther, 1999a) is employed. Its key idea is to allow incremental assignment of syllable roles to segmental positions via a featural decomposition of the three traditional roles, using two binary-valued features `ons` and `cod`:

(4)

Onset	ons	~cod
Nucleus	~ons	~cod
Coda	~ons	cod
Coda**O**nset	ons	cod

As a side-effect, one gets the fourth role **CO**, a monosegmental prosodic representation of true geminates. The subcomponent `sbs`, for sonority-based syllabification, itself rests on the computation of `sonority_differences` between adjacent segmental positions (not shown), where sonority may either go `up` or `down`. Together with some self-explanatory constraints `obligatory_wordinternal_onsets` and `no_geminates`, prosodic surface wellformedness is then welldefined. Only `if_doubly_synced_edge_then_stressed` may seem slightly odd, since it has a purely technical character: it rules out certain illformed alternatives in wordforms. Note, however, that the necesssity of such technical constraints, which are certainly implicit in informal analyses as well, can only be reliably detected in computerized analyses such as the present one, which allow for mechanical enumeration of a grammar's denotation.

```
sbs := [ {  [consumer(down&~ons),
             consumer(segment&~'Nuc')],
            [consumer(up&~'Nuc'),
             consumer(segment&~cod)
         } *, no_final_onset ^].

no_initial_coda := consumer(segment&~cod).
no_final_onset := consumer(segment&~ons).

syllabification := sonority_differences&
 sbs&[no_initial_coda, sbs].

% -- further constraints ---
obligatory_wordinternal_onsets :=
   ( segment -r-> ons / 'Nuc' ). % _ 'N'

no_geminates := consumer(~'CO')*.

prosodic_constraints := obligatory_word-
 internal_onsets & no_geminates &
 if_doubly_synced_edge_then_stressed.

if_doubly_synced_edge_then_stressed :=
 [( {consumer(~':1'),
    [consumer(':1'),consumer(~':1')],
    [consumer(':1'),consumer(':1'),
     consumer(stressed)]
    } *), consumer(':1') ^].
```

Stress Given the assignment of syllable roles to segmental positions, we are now ready to define Temiar word `stress`. A possibly empty sequence of `prefinal_syllables`, each of which is constrained to be of shape $ON(C)$ and `unstressed`, is followed by a final `stressed syllable`. The macro `ends_before_last_syll` makes sure that the dividing line between the penultimate and ultimate syllable is drawn correctly.

```
stress := [prefinal_syllables &
            ends_before_last_syll,
            syllable].

prefinal_syllables :=
   ([consumer('Ons'), consumer('Nuc'),
    (consumer('Cod') ^) ]*) &
    consumer(unstressed)*.

ends_before_last_syll:=([consumer(segment)*,
                   consumer(segment&~ons)]^).

syllable := [consumer(ons)+,consumer('Nuc'),
             consumer(cod)*] &
            (consumer(stressed)*).
```

Stems We proceed towards the definition of a **stem** by noting that – as described in §2 – both the extent of a **base**'s phonological material *and* its stress pattern are necessary prior knowledge for adding aspectual morphemes in the appropriate way. Hence, we impose the respective constraints onto the *isolated base string* in **stem0**, before wrapping the result in the usual enrichments. However, the addition of self loops for infixation this time is *a priori* restricted to the position immediately before a stressed onset, in accordance with the descriptive generalization stated in §2. Experiments have shown that using the unrestricted add_self_loops of (3.c) would cause much unnecessary hassle in *a posteriori* restriction of the possible infix locations to the actually attested ones. It thus appears that Temiar provides a first case for further parametrization of at least one of the original operators from (Walther, 1999b):

```
base := [consumer(':1'),consumer(':0')*,
         consumer(':1')].

stem0(StemMaterial) :=
 add_self_loop_before(stressed&'Ons',
  add_repeats(add_skips(StemMaterial &
   base & syllabification &
   prosodic_constraints & stress))).

stem(Segments) :=
 stem0(stringToSegments(Segments)).
```

Definitions for the actual stem entries of **selog, koow, yaap** are shown below, using the ASCII-IPA mapping {@ ↦ ə, E ↦ ɛ, O ↦ ɔ}. In evaluating the first entry, the schwa actually translates into a producer-type disjunction (**ə;ɛ**) with the help of **stringToSegments**. It thus makes sense to constrain this free alternation further, which is the purpose of **has_prefinal_syllable**. While the monosyllable **koow** needs no extra treatment, **yaap** is an example of a stem ending in an **alternating_labial**, whose definition however is straightforward (**medial, final** refer to a positional classification of the word that is defined later):

```
selog := stem("s@lOg") &
       has_prefinal_syllable.

koow := stem("kOOw").
yaap := stem0([stringToSegments("yaa"),
              alternating_labial]).

alternating_labial := {producer(p&final),
                   producer(m&medial&cod)}.
```

If we now define **has_prefinal_syllable** itself, we have completed the components that make up **stem**. While the definition really targets the prefinal vowel, its preceding onset and the stretch of arbitrary material after it must also be mentioned. To tolerate interspersed **technical_symbols**, the **ignore** operator is used (Kaplan and Kay, 1994).

The purpose of **prefinal_V** is to control the alternation between 'outer' and 'inner' vowel, here parametrized for ɛ∼ə only. It does so by referencing the next syllable role: if it is consistent with **ons**, that vowel resides in an open syllable, hence the **close_mid** variant (ə) will be selected. Two elsewhere cases deal with closed syllables and the possible presence of a technical symbol:

```
has_prefinal_syllable :=
  ignore([consumer('Ons'),
          prefinal_V(('E';'@'),
              ':0'&unstressed),
          consumer(anything) *],
          technical_symbols).

technical_symbols :=
 (consumer((skip;repeat)) *).

prefinal_V(Quality, Common) :=
 { [producer(Quality&close_mid&Common),
    consumer(ons)],
   [producer(Quality&~close_mid&Common),
    consumer(cod)],
   [consumer((skip;repeat))]
 } ).
```

Aspectual affixes It is time to concentrate on the most interesting part, and that is how to define the affixes. Again the general picture will be to see them as constraints on word forms which are imposed by intersection. We begin with the **simulfactive**. The claim here is that its characteristic pattern is the realization of the initial base segment (**:1**), followed by the infixed melodic element /a/, and then the entire string that begins with the stressed onset. Phrasing the pattern this way already suffices to capture the difference in reduplication behaviour between *ˈkɔɔw* and *səˈlɔg*: if we have inserted the *-a-* after the initial consonant in the first base, the stressed onset is *to the left of /a/'s position*, whereas in the second base that onset is found *to the right*. Thus, repetition of segments is necessary to avoid ungrammaticality due to constraint violation in the first case (*k-a-ˈkɔɔw*), but not in the second (*s-a-ˈlɔg*).

This behaviour is most naturally modelled by defining a new operator **seek(X)**, which allows for

ambiguous movement *either* to the left (`repeat`) *or* to the right (`skip`) before imposing the restriction `X`. This operator is applied to infixal /a/ because it is precisely the infix which needs to 'seek' its prosodically defined unique insertion point, i.e. self loop. Finally, to ensure that the other aspectual morphemes can play their part later on, the entire pattern is wrapped in `align` to tolerate further material before (`align_right`) and after it (`align_left`):

```
simulfactive :=
  align([consumer(':1'),
         seek([producer(a&':0'&unstressed),
         consumer(stressed&'Ons')])]).

seek(X) :=
 [{producer(skip)*,producer(repeat)*},X].

align_left(X):=[X,consumer(anything)*].
align_right(X):=[consumer(anything)*,X].
align(X) := align_right(align_left(X)).
```

Moving on to the `continuative`, we can see that the relevant formal generalization is a bit more complex. Again we start off with the initial base segment (`:1`), but then seek a place to infix the constant /ɛ/, before we `skip_to` the next synchronised base position (`:1`), which inevitably will be the final one. The pattern is completed by again seeking the stressed onset, from which realization of the string proceeds uninterrupted due to the licensing of extra material that the `align` wrapper provides. This produces a similar contrast with respect to (non-)reduplication of the first base position, but makes both the repetition of the last base segment and the ~~truncation~~ of its interior material obligatory in both base types (*k-ɛ- ~~oo~~ w'kɔɔw* vs. *s-ɛ- ~~lo~~ g'lɔg*):

```
continuative :=
 align([consumer(':1'),
    seek([producer('E'&':0'&unstressed)]),
       skip_to(consumer(':1')),
       seek(consumer(stressed&'Ons'))]).

skip_to(X) := [producer(skip)+, X].
```

What is left now is the proper definition of the `causative`. Here we observe from (1) that the causative morphology always starts word-initial, hence the use of `align_left`. We have a default consonant /t/ whose realization we must somehow force in the monosyllabic roots. Next comes a vowel, whose quality – ə or ɛ – is again regulated by the familiar `has_prefinal_syllable`. Finally, the characteristic fixed element /r/ is specified. Upon second thought, the /t/ is guaranteed to appear in monosyllable roots, because prefinal syllables always require an onset. The default absence of the /t/ – when not needed on prosodic grounds – is again encoded by the producer/consumer distinction, which contrasts the two disjuncts of the parametrized macro `default`:

```
causative :=
 align_left([default(t&unstressed,':1'),
            producer(vowel),
            producer(r&':1'&unstressed)])&
           has_prefinal_syllable.

default(Optional, Common) :=
   { producer(Common&Optional),
     consumer(Common) }.
```

Entire words We can put the pieces together now by first defining the `word` constraint as the conjunction of syllabification and related prosodic constraints plus a classification of the word's segmental positions into `initial,medial,final` ones. Again, this is modulo interspersed $repeat$ or $skip$ symbols. This actually means that base syllabification and word syllabification must match up, but fortunately this is indeed a property of our Temiar data.

Second, `wordform` conjoins the previous constraint with its parameter `X` – which will contain the conjunction of stem and aspect morphemes –, before eliminating leftover consumer symbols with the help of `closed_interpretation`:

```
word := ignore(syllabification &
               prosodic_constraints &
               positional_classification,
               technical_symbols).

positional_classification :=
[consumer(initial),consumer(medial)*,
 consumer(final)].

wordform(X):=closed_interpretation(X&word).
```

These definitions have removed the last barrier to evaluating expressions like `wordform(selog & simulfactive & causative)` or even suitable disjunctive combinations of such expressions which define entire paradigms. Figure 3 shows an example automaton for three forms. We refrain from describing a final automaton operation called Bounded Local Optimization in (Walther, 1999b) that was put

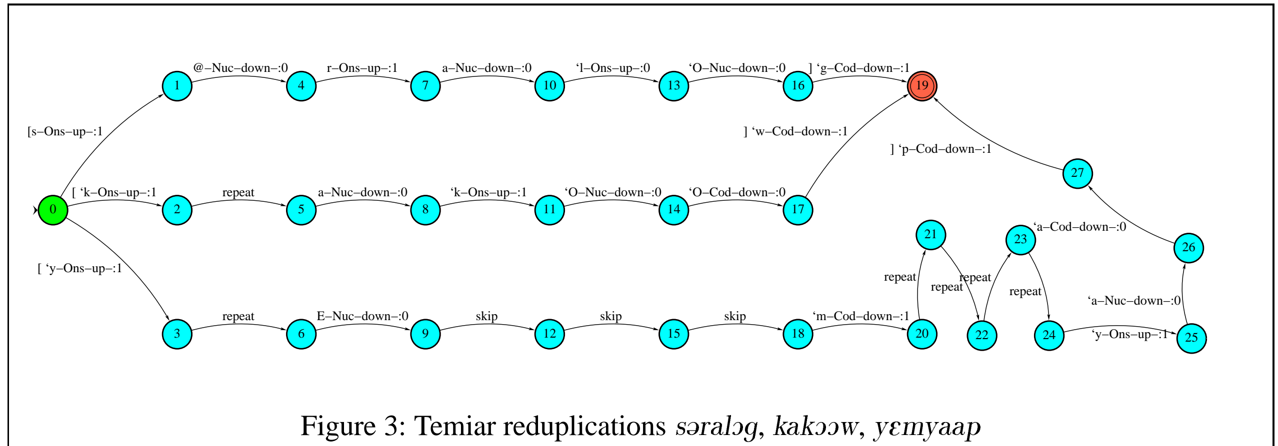


Figure 3: Temiar reduplications *səralɔg*, *kakɔɔw*, *yɛmyaap*

to use here to filter harmless spurious ambiguities from the original version of fig. 3. The kind of ambiguity involved in our Temiar grammar is one of alternative distribution of technical symbols in strings of the same segmental-content yield. Suffice to say that a simple parametrization of Bounded Local Optimization, which could only look at length-1 transition paths emerging from any given state, was able to prune the unwanted alternatives by considering technical transitions costlier in weight than segmental transitions.

5 Conclusion

The present paper has provided further support for (Walther, 1999b)'s finite-state conception of One-Level Prosodic Morphology by formulating – for the first time – a fully formalized and computational analysis of a complicated piece of reduplicative morphology found in the Mon-Khmer language of Temiar. Compared to the initial proposal, all three core components of enriched representations, namely technical transitions for repeating or skipping segmental symbols and the ability to perform infixation by using self loops, were again found necessary in the course of this analysis. However, in Temiar the last enrichment – add_self_loops – needed to be parametrized for a prosodic condition to narrow down the insertion site to a unique position per base.

The prosodic condition of 'stressed onset' proved crucial to define that position, and accounted for the variation between infixing aspectual morphology in longer bases and descriptively prefixing morphology in monosyllabic ones. Temiar thus underscores the utility of computing with real prosodic information in finite-state morphology, a frequently missing desideratum according to (Sproat, 1992, p.170). Also, the symmetry of having both forward and backward-pointing technical transitions in enriched automata representations was exploited in a novel regular expression operator called `seek(X)`, which encapsulated an interesting kind of ambiguous directional movement (or: movement underspecified for direction) towards a position satisfying property `X`. This operator could rather directly be motivated from the data. In particular, it facilitated an insightful account of the base-length-dependent triggering of reduplication in the active simulfactive aspect.

Finally, in contrast to even the most recent analyses in the theoretical linguistic literature, the full paradigm including the causative forms was captured in this fairly complete analysis, together with phonological modifications that sometimes occur between base and reduplicant, as exemplified by *yɛ***m***yaa***p**. Apart from an optional filtering step for some technical spurious ambiguities that could make use of local optimization, neither global optimization nor violable or soft constraints of the type argued for in Optimality Theory (Prince and Smolensky, 1993) were found necessary.

For future research, the empirical base of Temiar should be broadened to include further reduplication patterns, in particular those found in expressives. Also, the grammar should be amended to allow for words containing geminates, which were initially excluded to simplify the overall analysis at the cost of what is at best a peripheral aspect of it. Because the finite-state constraints employed in this work are all surface-true, the potential of machine-learning techniques to acquire them automatically from surface-oriented corpora should be explored. Finally, it would be very interesting to broaden to Temiar the ongoing experiments with efficiency-oriented computational variants of the One-Level Prosodic Morphology framework that were already alluded to in the text.

References

Evan Antworth. 1990. *PC-KIMMO: A Two-Level Processor for Morphological Analysis*. SIL, Dallas.

Kenneth R. Beesley. 1998. Constraining separated morphotactic dependencies in finite-state grammars. In *Proceedings of FSMNLP'98, Bilkent University, Turkey*, pages 118–127.

Geoffrey Benjamin. 1976. An outline of Temiar grammar. In Philip Jenner, Lawrence Thompson, and Stanley Starosta, editors, *Austroastiatic studies*, volume II, pages 129–187. University Press of Hawaii, Honululu.

Steven Bird and T. Mark Ellison. 1992. One-Level Phonology: Autosegmental representations and rules as finite-state automata. Technical report, Centre for Cognitive Science, University of Edinburgh. EUCCS/RP-51.

Steven Bird and T. Mark Ellison. 1994. One-Level Phonology. *Computational Linguistics*, 20(1):55–90.

Ellen Broselow and John McCarthy. 1983. A theory of infixing reduplication. *The Linguistic Review*, 3:25–98.

Adamantios Gafos. 1995. On the Proper Characterization of 'Nonconcatenative' Languages. Ms., Department of Cognitive Science, The Johns Hopkins University, Baltimore. (ROA-106 at http://ruccs.rutgers.edu/roa.html).

Diamandis Gafos. 1996. *The articulatory basis of locality in phonology*. Ph.D. thesis, The Johns Hopkins University, Baltimore, Md. [Published by Garland:New York].

Diamandis Gafos. 1998a. A-templatic reduplication. *Linguistic Inquiry*, 29(3):515–527.

Diamandis Gafos. 1998b. Eliminating long distance consonantal spreading. *Natural Language and Linguistic Theory*, 16(2):223–278.

Ron Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–78.

John McCarthy. 1982. Prosodic templates, morphemic templates, and morphemic tiers. In Harry van der Hulst and Norval Smith, editors, *The structure of phonological representations, part I*, pages 191–224. Foris, Dordrecht.

Alan Prince and Paul Smolensky. 1993. Optimality theory. constraint interaction in generative grammar. Technical Report RuCCS TR-2, Rutgers University Center for Cognitive Science.

Patricia Shaw. 1993. The prosodic constituency of minor syllables. In *Proceedings of the Eleventh West Coast Conference on Formal Linguistics*, pages 117–132, Stanford, CA. CSLI Publications. [Distributed by Cambridge University Press].

Kelly Sloan. 1988. Bare-consonant reduplication. In *Proceedings of the Seventh West Coast Conference on Formal Linguistics*, pages 319–330, Stanford, CA. CSLI Publications. [Distributed by Cambridge University Press].

Richard Sproat. 1992. *Morphology and Computation*. MIT Press, Cambridge, Mass.

Gertjan van Noord. 1997. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derrick Wood, and Sheng Yu, editors, *Automata Implementation*, volume 1260 of *Lecture Notes in Computer Science*, pages 87–108. Springer Verlag. (Software under `http://grid.let.rug.nl/~vannoord/Fsa/`).

Markus Walther. 1999a. *Deklarative prosodische Morphologie: constraint-basierte Analysen und Computermodelle zum Finnischen und Tigrinya*. Niemeyer, Tübingen.

Markus Walther. 1999b. One-Level Prosodic Morphology. Marburger Arbeiten zur Linguistik 1, University of Marburg. 64 pp. (http://xxx.lanl.gov/abs/cs.CL/9911011).

Markus Walther. 2000. Finite-state Reduplication in One-Level Prosodic Morphology. In *Proceedings of NAACL-2000*, pages 296–302, Seattle/WA. North American Association for Computational Linguistics, Morgan Kaufman. (http://xxx.lanl.gov/abs/cs.CL/0005025).

Easy and Hard Constraint Ranking in Optimality Theory:[*]
Algorithms and Complexity

Jason Eisner
Dept. of Computer Science / University of Rochester
Rochester, NY 14607-0226 USA / jason@cs.rochester.edu

Abstract

We consider the problem of ranking a set of OT constraints in a manner consistent with data. (1) We speed up Tesar and Smolensky's RCD algorithm to be linear on the number of constraints. This finds a ranking such that each attested form x_i beats or ties a particular competitor y_i. (2) We generalize RCD so each x_i beats or ties *all* possible competitors. (3) Alas, if the surface form of x_i is only partially observed, then an NP-hardness construction suggests that it is effectively necessary to consider all possible rankings or surface forms. (4) Merely checking that a *single* (given) ranking is consistent with data is coNP-complete if the surface forms are fully observed and Δ_2^p-complete if not (since OT generation is OptP-complete).(5) Determining whether *any* consistent ranking exists is coNP-hard (but in Δ_2^p) if the surface forms are fully observed, and Σ_2^p-complete if not. (6) Generation (P) and ranking (NP-complete) in derivational theories are easier than in OT.

1 Introduction

Optimality Theory (OT) is a grammatical paradigm that was introduced by Prince and Smolensky (1993) and suggests various computational questions, including learnability.

Following Gold (1967) we might ask: Is the language class $\{L(\mathcal{G}) : \mathcal{G}$ is an OT grammar$\}$ learnable in the limit? That is, is there a learning algorithm that will converge on any OT-describable language $L(\mathcal{G})$ if presented with an enumeration of its grammatical forms?

In this paper we consider an orthogonal question that has been extensively investigated by Tesar and Smolensky (1996), henceforth T&S. Rather than asking whether a learner can eventually find an OT grammar compatible with an unbounded set of positive data, we ask: How efficiently can it find a grammar (if one exists) compatible with a finite set of positive data?

We will consider successively more realistic versions of the problem, as described in the abstract. The easiest version turns out to be easier than previously known. The harder versions turn out to be harder than previously known.

2 Formalism

An OT grammar $\mathcal{G}$ consists of three elements, any or all of which may need to be learned:

- a set $\mathcal{L}$ of **underlying forms** produced by a lexicon or morphology,
- a function Gen that maps any underlying form to a set of **candidates**, and
- a vector $\vec{C} = \langle C_1, C_2, \ldots C_n \rangle$ of **constraints**, each of which is a function from candidates to natural numbers.

C_i is said to **rank** higher than (or **outrank**) C_j in $\vec{C}$ iff $i < j$. We say x **satisfies** C_i if $C_i(x) = 0$, else x **violates** C_i.

The grammar $\mathcal{G}$ defines a relation that maps each $u \in \mathcal{L}$ to the candidate(s) $x \in \mathsf{Gen}(u)$ for which the vector $\vec{C}(x) \stackrel{\text{def}}{=} \langle C_1(x), C_2(x), \ldots C_n(x) \rangle$ is lexicographically minimal. Such candidates are called **optimal**.

One might then say that the grammatical forms are the pairs (u, x) of this relation. But for simplicity of notation and without loss of generality, we will suppose that the candidates x are rich enough that u can always be recovered from x.[1] Then u is redundant and we may simply take the candidate x to be the grammatical form. Now the language $L(\mathcal{G})$ is simply the image of $\mathcal{L}$ under $\mathcal{G}$. We will write u_x for the underlying form, if any, such that $x \in \mathsf{Gen}(u_x)$.

An **attested form** of the language is a candidate x that the learner knows to be grammatical (i.e., $x \in L(\mathcal{G})$). y is a **competitor** of x if they are both in the same candidate set: $u_x = u_y$. If x, y are competitors with $\vec{C}(y) < \vec{C}(x)$, we say that y **beats** x (and then x is not optimal).

[*] Many thanks go to Lane and Edith Hemaspaandra for references to the complexity literature, and to Bruce Tesar for comments on an earlier draft.

[1] This is necessary in any case if the constraints $C_j(x)$ are to depend on (all of) u. In general, we expect that each candidate $x \in \mathsf{Gen}(u)$ encodes an alignment of the underlying form u with some possible surface form s, and $C_j(x)$ evaluates this *pair* on some criterion.

An ordinary learner does not have access to attested forms, since observing that $x \in L(\mathcal{G})$ would mean observing an utterance's entire prosodic structure and underlying form, which ordinarily are not vocalized. An **attested set** of the language is a set X such that the learner knows that some $x \in X$ is grammatical (but not necessarily *which* x). The idea is that a set is attested if it contains all possible candidates that are consistent with something a learner heard.[2] An **attested surface set**—the case considered in this paper—is an attested set all of whose elements are competitors; i.e., the learner is sure of the underlying form but not the surface form.

Some computational treatments of OT place restrictions on the grammars that will be considered. The **finite-state assumptions** (Ellison, 1994; Eisner, 1997a; Frank and Satta, 1998; Karttunen, 1998; Wareham, 1998) are that

- candidates and underlying forms are represented as strings over some alphabet;
- Gen is a regular relation;[3]
- each C_j can be implemented as a weighted deterministic finite-state automaton (WDFA) (i.e., $C_j(x)$ is the total weight of the path accepting x in the WDFA);
- $\mathcal{L}$ and any attested sets are regular.

The **bounded-violations assumption** (Frank and Satta, 1998; Karttunen, 1998) is that the value of $C_j(x)$ cannot increase with $|x|$, but is bounded above by some k.

In this paper, we do not always impose these additional restrictions. However, when demonstrating that problems are hard, we usually adopt both restrictions to show that the problems are hard even for the restricted case.

Throughout this paper, we follow T&S in supposing that the learner already knows the correct *set* of constraints $\mathcal{C} = \{C_1, C_2, \ldots C_n\}$, but must learn their order $\vec{C} = \langle C_1, C_2, \ldots C_n \rangle$, known as a **ranking** of $\mathcal{C}$. The assumption follows from the OT philosophy that $\mathcal{C}$ is universal across languages, and only the order of constraints differs. The algorithms for learning a ranking, however, are designed to be general for any $\mathcal{C}$, so they take $\mathcal{C}$ as an input.[4]

3 RCD as Topological Sort

T&S investigate the problem of ranking a constraint set $\mathcal{C}$ given a set of attested forms $x_1, \ldots x_m$ and corresponding competitors $y_1, \ldots y_m$. The problem is to determine a ranking $\vec{C}$ such that for each i, $\vec{C}(x_i) \leq \vec{C}(y_i)$ lexicographically. Otherwise x_i would be ungrammatical, as witnessed by y_i.

In this section we give a concise presentation and analysis of T&S's **Recursive Constraint Demotion (RCD)** algorithm for this problem. Our presentation exposes RCD's connection to topological sort, from which we borrow a simple bookkeeping trick that speeds it up.

3.1 Compiling into Boolean Formulas

The first half of the RCD algorithm extracts the relevant information from the $\{x_i\}$ and $\{y_i\}$, producing what T&S call *mark-data pairs*. We use a variant notation. For each constraint $C \in \mathcal{C}$, we construct a negation-free, conjunctive-normal form (CNF) Boolean formula $\phi(C)$ whose literals are other constraints:

$$\phi(C) = \bigwedge_{i:C(x_i)>C(y_i)} \; \bigvee_{C':C'(x_i)<C'(y_i)} C'$$

[2]This is of course a simplification. Attested sets corresponding to *laugh* and *laughed* can represent the learner's uncertainty about the respective underlying forms, but not the knowledge that the underlying forms are *related*. In this case, we can solve the problem by packaging the entire morphological paradigm of *laugh* as a single candidate, whose attested set is constrained by the two surface observations *and* by the requirement of a shared underlying stem. (A k-member paradigm may be encoded in a form suitable to a finite-state system by interleaving symbols from $2k$ aligned tapes that describe the k underlying and k surface forms.) Alas, this scheme only works within disjoint finite paradigms: while it captures the shared underlying stem of *laugh* and *laughed*, it ignores the shared underlying *suffix* of *laughed* and *frowned*.

[3]Ellison (1994) makes only the weaker assumption that $\mathsf{Gen}(u)$ is a regular set for each u.

[4]Again this is an oversimplification. Given the variety of constraints already proposed in the phonological literature, $n = |\mathcal{C}|$ would have to be extremely large for $\mathcal{C}$ to include all possible cross-linguistic constraints. The methods here are probably impractical for such large n, since they are designed to work on arbitrary $\mathcal{C}$ and therefore spend some time on each constraint separately, if only to read it from the input. An alternative would be to tailor an algorithm to a particular constraint set $\mathcal{C}$, making it possible to exploit that set's internal structure. Consider e.g. Eisner's proposal (1997b; 1997a) that $\mathcal{C}$ is the union of two simple parametric constraint families. Note that such an algorithm would not have time to output a total ranking of $\mathcal{C}$ by enumeration; it might output a concise representation of the ranking, or a short prefix $C_1, \ldots C_k$ that is sufficient to determine which forms (or which attested forms) are grammatical.

The interpretation of the literal C' in $\phi(C)$ is that C' is ranked before C. It is not hard to see that a constraint ranking is a valid solution iff it satisfies $\phi(C)$ for every C. For example, if $\phi(d) = (a \vee b \vee c) \wedge (b \vee e \vee f)$, this means that d must be outranked by either a, b or c (else x_1 is ungrammatical) and also by either b, e or f (else x_2 is ungrammatical).

How expensive is this compilation step? Observe that the inner term $\bigvee_{C':C'(x_i)<C'(y_i)} C'$ is independent of C, so it only needs to be computed and stored once. Call this term D_i. We first construct all m of the disjunctive clauses D_i, requiring time and storage $O(mn)$. Then we construct each of the n formulas $\phi(C) = \bigwedge_{i:C(x_i)>C(y_i)} D_i$ as a list of pointers to up to m clauses, again taking time and storage $O(mn)$.

The computation time is $O(mn)$ for the steps we have already considered, but we must add $O(mnE)$, where E is the cost of precomputing each $C(x_i)$ or $C(y_i)$ and may depend on properties of the constraints and input forms.

We write $M(= O(mn))$ for the exact storage cost of the formulas, i.e., $M = \sum_i |D_i| + \sum_C |\phi(C)|$ where $|\phi(C)|$ counts only the number of conjuncts.

3.2 Finding a Constraint Ranking

The problem is now to find a constraint ranking that satisfies $\phi(C)$ for every $C \in \mathcal{C}$. Consider the special case where each $\phi(C)$ is a simple conjunction of literals—that is, $(\forall i)|D_i| = 1$. This is precisely the problem of topologically sorting a directed graph with n vertices and $\sum_C |\phi(C)| = M/2$ edges. The vertex set is $\mathcal{C}$, and $\phi(C)$ lists the parents of vertex C, which must all be enumerated before C.

Topological sort has two well-known $O(M + n)$ algorithms (Cormen et al., 1990). One is based on depth-first search. Here we will focus on the other, which is: Repeatedly find a vertex with no parents, enumerate it, and remove it and its outgoing edges from the graph.

The second half of T&S's RCD algorithm is simply the obvious generalization of this topological sort method (to directed hypergraphs, in fact, formally speaking). We describe it as a function $\textsc{Rcd}(\mathcal{C}, \phi)$ that returns a ranking $\vec{C}$:

1. If $\mathcal{C} = \emptyset$, return $\langle\rangle$. Otherwise:
2. Identify a $C_1 \in \mathcal{C}$ such that $\phi(C_1)$ is empty. (C_1 is surface-true, or "undominated.")
3. If there is no such constraint, then fail: no ranking can be consistent with the data.
4. Else, for each $C \in \mathcal{C}$, destructively remove from $\phi(C)$ any disjunctive clause D_i that mentions C_1.
5. Now recursively compute and return $\vec{C} = \langle C_1, \textsc{Rcd}(\mathcal{C} - \{C_1\}, \phi)\rangle$.

Correctness of $\textsc{Rcd}(\mathcal{C}, \phi)$ is straightforward, by induction on $n = |\mathcal{C}|$. The base case $n = 0$ is trivial. For $n > 0$: $\phi(C_1)$ is empty and therefore satisfied. $\phi(C)$ is also satisfied for all other C: any clauses containing C_1 are satisfied because C_1 outranks C, and any other clauses are preserved in the recursive call and therefore satisfied by the inductive hypothesis.

We must also show completeness of $\textsc{Rcd}(\mathcal{C}, \phi)$: if there exists at least one correct answer $\vec{B}$, then the function must not fail. Again we use induction on n. The base case $n = 0$ is trivial. For $n > 0$: Observe that $\phi(B_1)$ is satisfied in $\vec{B}$, by correctness of $\vec{B}$. Since B_1 is not outranked by anything, this implies that $\phi(B_1)$ is empty, so $\textsc{Rcd}$ has at least one choice for C_1 and does not fail. It is easy to see that $\vec{B}$ with C_1 removed would be a correct answer for the recursive call, so the inductive hypothesis guarantees that that call does not fail either.

3.3 More Efficient Bookkeeping

T&S (p. 61) analyze the $\textsc{Rcd}$ function as taking time $O(mn^2)$; in fact their analysis shows more precisely $O(Mn)$. We now point out that careful bookkeeping can make it operate in time $O(M + n)$, which is at worst $O(mn)$ provided $n > 0$. This means that the whole RCD algorithm can be implemented in time $O(mnE)$, i.e., it is bounded by the cost of applying all the constraints to all the forms.

First consider the special case discussed above, topological sort. In linear-time topological sort, each vertex maintains a list of its children and a count of its parents, and the program maintains a list of vertices whose parent count has become 0. The algorithm then requires only $O(1)$ time to find and remove each vertex, and $O(1)$ time to remove each edge, for a total time of $O(M + n)$ plus $O(M + n)$ for initialization.

We can organize RCD similarly. We change our representations (not affecting the compilation time in §3.1). Constraint C need not

store $\phi(C)$. Rather, C should maintain a list of pointers to clauses D_i in which it appears as a disjunct (cf. "a list of its children") as well as the integer $|\phi(C)|$ (cf. "a count of its parents"). The program should maintain a list of "undominated" constraints for which $|\phi(C)|$ has become 0. Finally, each clause D_i should maintain a list of constraints C such that D_i appears in $\phi(C)$.

Step 2 of the algorithm is now trivial: remove the head C_1 of the list of undominated constraints. For step 4, iterate over the stored list of clauses D_i that mention C_1. Eliminate each such D_i as follows: iterate over the stored list of constraints C whose $\phi(C)$ includes D_i (and then reset that list to empty), and for each such C, decrement $|\phi(C)|$, adding C to the undominated list if $|\phi(C)|$ becomes 0.

The storage cost is still $O(M+n)$. In particular, $\phi(C)$ is now implicitly stored as $|\phi(C)|$ backpointers from its clauses D_i, and D_i is now implicitly stored as $|D_i|$ backpointers from its disjuncts (e.g., C_1). Since RCD removes each constraint and considers each backpointer exactly once, in $O(1)$ time, its runtime is $O(M+n)$.

In short, this simple bookkeeping trick eliminates RCD's quadratic dependence on n, the number of constraints to rank. As already mentioned, the total runtime is now dominated by $O(mnE)$, the preprocessing cost of applying all the constraints to all the input forms. Under the finite-state assumption, this can be be more tightly bounded as $O(n \cdot \text{total size of input forms}) = O(n \cdot \sum_i |x_i| + |y_i|)$, since the cost of running a form through a WDFA is proportional to the former's length.

3.4 Alternative Algorithms

T&S also propose an alternative to RCD called Constraint Demotion (CD), which is perhaps better-known. (They focus primarily on it, and the textbook of (Kager, 1999) devotes a chapter to it.) A disjunctive clause D_i (compiled as in §3.1) is **processed** roughly as follows: for each C such that D_i is an unsatisfied clause of $\phi(C)$, greedily satisfy it by demoting C as little as possible. CD repeatedly processes $D_1, \ldots D_m$ until all clauses in all formulas are satisfied.

CD can be efficiently implemented so that each pass through all clauses takes time proportional to M. But it is easy to construct datasets that require $n+1$ passes. So the ranking step can take time $\Omega(Mn)$, which contrasts unfavorably with the $O(M+n)$ time for RCD.

CD does have the nice property (unlike RCD) that it maintains a constraint ranking at all times. An "online" (memoryless) version of CD is simply to generate, process, and discard each clause D_i upon arrival of the new data pair x_i, y_i; this converges, given sufficient data. But suppose one wishes to maintain a ranking that is consistent with *all* data seen so far. In this case, CD is slower than RCD. Modifying a previously correct ranking to remain correct given the new clause D_i requires at least one pass through all clauses $D_1, \ldots D_i$ (as slow as RCD) and up to $n+1$ passes (as slow as running CD on all clauses from scratch, ignoring the previous ranking).

4 Considering *All* Competitors

The algorithms of the previous section only ensure that each attested form x_i is at least as harmonic as a given competitor y_i: $\vec{C}(x_i) \leq \vec{C}(y_i)$. But for x_i to be grammatical, it must be at least as harmonic as *all* competitors. We would like a method that ensures this. Such a method will rank a constraint set $\mathcal{C}$ given only a set of attested forms $\{x_1, \ldots x_m\}$.

Like T&S, whose algorithm for this case is discussed in §4.2, we will assume that we have an efficient computation of the OT generation function $\text{OPT}(\vec{C}, u)$. (See e.g. (Ellison, 1994; Tesar, 1996; Eisner, 1997a).) This returns the subset of $\mathsf{Gen}(u)$ on which $\vec{C}(\cdot)$ is lexicographically minimal, i.e., the set of grammatical outputs for u. For purposes of analysis, we let P be a bound on the runtime of our OPT algorithm. We will discuss this runtime further in §6.

4.1 Generalizing RCD

We propose to solve this problem by running something like our earlier RCD algorithm, but considering all competitors at once.

First, as a false start, let us try to construct the requirements $\phi(C)$ in this case. Consider the contribution of a single x_i to a particular $\phi(C)$. x_i demands that for *any* competitor y such that $C(x_i) > C(y)$, C must be outranked by *some* C' such that $C'(x_i) < C'(y)$. One set of competitors y might all add the same clause $(a \vee b \vee c)$ to $\phi(C)$; another set might add a different clause $(b \vee d \vee e)$.

The trouble here is that $\phi(C)$ may become intractably large. This will happen if the con-

straints are roughly orthogonal to one another. For example, suppose the candidates are bit strings of length n, and for each k, there exists a constraint $\textsc{Off}_k$ preferring the kth bit to be zero.[5] If $x_i = 1000\cdots0$, then $\phi(\textsc{Off}_1)$ contains all 2^{n-1} possible clauses: for example, it contains $(\textsc{Off}_2 \vee \textsc{Off}_4 \vee \textsc{Off}_5)$ by virtue of the competitor $y = 0101100000\cdots$. Of course, the conjunction of all these clauses can be drastically simplified in this case, but not in general.

Therefore, we will skip the step of constructing formulas $\phi(C)$. Rather, we will run something like RCD directly: greedily select a constraint C_1 that does not eliminate any of the attested forms x_i (but that may eliminate some of its competitors), similarly select C_2, etc.

In our new function $\textsc{RcdAll}(\mathcal{C}, \vec{B}, \{x_i\})$, the input includes a partial hierarchy $\vec{B}$ listing the constraints chosen at previous steps in the recursion. (On a non-recursive call, $\vec{B} = \langle\rangle$.)

1. If $\mathcal{C} = \emptyset$, return $\langle\rangle$. Otherwise:
2. By trying all constraints, find a constraint C_1 such that $(\forall i) x_i \in \textsc{Opt}(\langle\vec{B}, C_1\rangle, u_{x_i})$
3. If there is no such constraint, then fail: no ranking can be consistent with the data.
4. Else recursively compute and return $\vec{C} = \langle C_1, \textsc{RcdAll}(\mathcal{C} - \{C_1\}, \langle\vec{B}, C_1\rangle, \{x_i\})\rangle$

It is easy to see by induction on $|\mathcal{C}|$ that RCDALL is correct: if it does not fail, it always returns a ranking $\vec{C}$ such that each x_i is grammatical under the ranking $\langle\vec{B}, \vec{C}\rangle$. It is also complete, by the same argument we used for RCD: if there exists a correct ranking, then there is a choice of C_1 for this call and there exists a correct ranking on the recursive call.

The time complexity of RCDALL is $O(mn^2P)$. Preprocessing and compilation are no longer necessary (that work is handled by OPT). We note that if OPT is implemented by successive winnowing of an appropriately represented candidate set, as is common in finite-state approaches, then it is desirable to cache the sets returned by OPT at each call, for use on the recursive call. Then $\textsc{Opt}(\langle\vec{B}, C_1\rangle, u_{x_i})$ need not be computed from scratch: it is simply the subset of $\textsc{Opt}(\vec{B}, u_{x_i})$ on which $C_1(\cdot)$ is minimal.

[5] $\textsc{Off}_k(x)$ simply extracts the kth bit of x. We will later denote it as $C_{\neg v_k}$.

4.2 Alternative Algorithms

T&S provide a different, rather attractive solution to this problem, which they call Error-Driven Constraint Demotion (EDCD). This is identical to the "online" CD algorithm of §3.4, except that for each attested form x that is presented to the learner, EDCD automatically chooses a competitor $y \in \textsc{Opt}(\vec{C}, u_x)$, where $\vec{C}$ is the ranking at the time.

If the supply of attested forms $x_1, \ldots x_m$ is limited, as assumed in this paper, one may iterate over them repeatedly, modifying $\vec{C}$, until they are all optimal. When an attested form x is suboptimal, the algorithm takes time $O(nE)$ to compile x, y into a disjunctive clause and time $O(n)$ to process that clause using CD.[6]

T&S show that the learner converges after seeing at most $O(n^2)$ suboptimal attested forms, and hence after at most $O(n^2)$ passes through $x_1, \ldots x_m$. Hence the total time is $O(n^3E + mn^2P)$, where P is the time required by OPT. This is superficially worse than our RCDALL, which takes time $O(mn^2P)$, but really the same since P dominates (see §6).

Nonetheless, §7 will discuss a genuine sense in which RCDALL is more efficient than EDCD (and MRCD), thanks to the more limited information it gets from OPT.

Algorithms that adjust constraint rankings or weights along a continuous scale include the Gradual Learning Algorithm (Boersma, 1997), which resembles simulated annealing, and maximum likelihood estimation (Johnson, 2000). These methods have the considerable advantage that they can deal with noise and free variation in the attested data. Both algorithms repeat until convergence, which makes it difficult to judge their efficiency except by experiment.

5 Incompletely Observed Forms

We now add a further wrinkle. Suppose the input to the learner specifies only $\mathcal{C}$ together with attested surface sets $\{X_i\}$, as defined in §2, rather than attested forms. This version of the problem captures the learner's uncertainty

[6] Instead of using CD on the new clause only, one may use RCD to find a ranking consistent with all clauses generated so far. This step takes worst-case time $O(n^2)$ rather than $O(n)$ even with our improved algorithm, but may allow faster convergence. Tesar (1997) calls this version Multi-Recursive Constraint Demotion (MRCD).

about the full description of the surface material. As before, the goal is to rank $\mathcal{C}$ in a manner consistent with the input.

With this wrinkle, even determining whether such a ranking exists turns out to be surprisingly harder. In §7 we will see that it is actually Σ_2^p-complete. Here we only show it NP-hard, using a construction that suggests that the NP-hardness stems from the need to consider exponentially many rankings or surface forms.

5.1 NP-Hardness Construction

Given n, we will be considering finite-state OT grammars of the following form:

- $\mathcal{L} = \{\epsilon\}$.
- $\mathsf{Gen}(\epsilon) = \Sigma^n$, the set of all length-$n$ strings over the alphabet $\Sigma = \{1, 2, \ldots n\}$. (This set can be represented with a straight-line DFA of $n+1$ states and n^2 arcs.)
- $\mathcal{C} = \{\textsc{Early}_j : 1 \leq j \leq n\}$, where for any $x \in \Sigma^*$, the constraint $\textsc{Early}_j(x)$ counts the number of digits in x before the first occurrence of digit j, if any. For example, $\textsc{Early}_3(2188353) = \textsc{Early}_3(2188) = 4$. (Each such constraint can be implemented by a WDFA of 2 states and $2n$ arcs.)

$\textsc{Early}_j$ favors candidates in which j appears early. The ranking $\langle \textsc{Early}_5, \textsc{Early}_8, \textsc{Early}_1, \ldots \rangle$ favors candidates of the form $581 \cdots$; no other candidate can be grammatical.

Given a directed graph G with n vertices identified by the digits $1, 2, \ldots n$. A **path** in G is a string of digits $j_1 j_2 j_3 \cdots j_k$ such that G has edges from j_1 to j_2, j_2 to j_3, ... and j_{k-1} to j_k. Such a string is called a **Hamilton path** if it contains each digit exactly once. It is an NP-complete problem to determine whether an arbitrary graph G has a Hamilton path.

Suppose we let the attested surface set X_1 be the set of length-n paths of G. This is a regular set that can be represented in space proportional to $n|G|$, by intersecting the DFA for $\mathsf{Gen}(\epsilon)$ with a DFA that accepts all paths of G.[7]

Now $(\mathcal{C}, \{X_1\})$ is an instance of the ranking problem whose size is $O(n|G|)$. We observe that any correct ranking algorithm succeeds iff G has a Hamilton path. Why? A ranking is a vector $\vec{C} = \langle \textsc{Early}_{j_1}, \ldots \textsc{Early}_{j_n} \rangle$, where $j_1, \ldots j_n$ is a permutation of $1, \ldots n$. The optimal form under this ranking is in fact the string $j_1 \cdots j_n$. A string is consistent with X_1 if it is a path of G, so the ranking $\vec{C}$ is consistent with X_1 iff $j_1 \ldots j_n$ is a Hamilton path of G. If such a ranking exists, the algorithm is bound to find it, and otherwise to return a failure code. Hence the ranking problem of this section is NP-hard.

[7]The latter DFA is isomorphic to G plus a start state. The states are $0, 1, \ldots n$; there is an arc from j to j' (labeled with j') iff $j = 0$ or G has an edge from j to j'.

5.2 Discussion

The NP-hardness effectively means that (unless P = NP) no general algorithm can always do better than checking each ranking or each possible surface form individually. This is not quite obvious, since in general, NP-hardness or coNP-hardness can also arise from the difficulty of checking whether a particular *one* of the surface forms is compatible with a particular *one* of the constraint rankings (see §6). However, that is not the case here, since the constraints $\textsc{Early}_j$ used in our construction interact in a simple and tractable way. (In particular, the winnowed candidate set after the first k constraints, $\textsc{Opt}(\langle \textsc{Early}_{j_1}, \ldots \textsc{Early}_{j_k} \rangle, \epsilon)$, is simply $j_1 \cdots j_k \Sigma^{n-k}$, a regular set that may be represented as a DFA of size $O(n^2)$.)

Note that our construction shows NP-hardness for even a restricted version of the ranking problem: finite-state grammars and finite attested surface sets. The result holds up even if we also make the bounded-violations assumption (see §2): the violation count can stop at n, since $\textsc{Early}_j$ need only work correctly on strings of length n. We revise the construction, modifying the automaton for each $\textsc{Early}_j$ by intersection (more or less) with the straight-line automaton for Σ^n. This enlarges the input to the ranking algorithm by a factor of $O(n)$.

By way of mitigating this stronger result, we note that the construction in the previous paragraph bounds $|X_i|$ by $n!$ and the number of violations by n. These bounds (as well as $|\mathcal{C}| = n$) increase with the order n of the input graph. If the bounds were imposed by universal grammar, the construction would not be possible and NP-hardness might not hold. Unfortunately, any universal bounds on $|X_i|$ or $|\mathcal{C}|$ would hardly be small enough to protect the ranking algorithm from having to solve huge instances of Hamilton

path.[8] As for bounded violations, the only real reason for imposing this restriction is to ensure that the OT grammar defines a regular relation (Frank and Satta, 1998; Karttunen, 1998). In recent work, Eisner (2000) argues that the restriction is too severe for linguistic description, and proposes a more general class of "directional constraints" under which OT grammars remain regular.[9] If this relaxed restriction is substituted for a universal bound on violations, the ranking problem remains NP-hard, since each EARLY_j is a directional constraint.

A more promising "way out" would be to universally restrict the size or structure of the automaton that describes the attested set. The set used in our construction was quite artificial.

However, in §7 we will answer all these objections: we will show the problem to be Σ_2^p-complete, using a natural attested set and binary-valued finite-state constraints (which, however, will not interact as simply).

5.3 Available Algorithms

The NP-hardness results above suggests that existing algorithms designed for this ranking problem are either incorrect or intractable on certain cases. Again, this does not rule out efficient algorithms for variants of the problem—see e.g. footnote 4—nor does it rule out algorithms that tend to perform well in the average case or on small inputs or on real data.

T&S proposed an algorithm for this problem, RIP/CD, but left its efficiency and correctness for future research (p. 39); Tesar and Smolensky (2000) show that it is not guaranteed to succeed. Tesar (1997) gives a related algorithm based on MRCD (see §4.2), but which sometimes requires iterating over all the candidates in an attested surface set; this might easily be intractable even when the set is finite.

6 Complexity of OT Generation

The ranking algorithms in §§4.1–4.2 relied on the existence of an algorithm to compute the independently interesting "language production" function $\text{OPT}(\vec{C}, u)$, which maps underlying u to the set of optimal candidates in $\mathsf{Gen}(u)$.

In this section, we consider the computational complexity of some functions related to OPT:[10]

- $\text{OPTVAL}(\vec{C}, u)$: returns $\min_{x \in \mathsf{Gen}(u)} \vec{C}(x)$. This is the violation vector shared by all the optimal candidates $x \in \text{OPT}(\vec{C}, u)$.
- $\text{OPTVALZ}(\vec{C}, u)$: returns "yes" iff the last component of the vector $\text{OPTVAL}(\vec{C}, u)$ is zero. This decision problem is interesting only because if it cannot be computed efficiently then neither can OPTVAL.
- $\text{BEATABLE}(\vec{C}, u, \langle k_1, \dots k_n \rangle)$: returns "yes" iff $\text{OPTVAL}(\vec{C}, u) < \langle k_1, \dots k_n \rangle$.
- $\text{BEST}(\vec{C}, u, \langle k_1, \dots k_n \rangle)$: returns "yes" iff $\text{OPTVAL}(\vec{C}, u) = \langle k_1, \dots k_n \rangle$.
- $\text{CHECK}(\vec{C}, x)$: returns "yes" iff $x \in \text{OPT}(\vec{C}, u_x)$. This checks whether an attested form is consistent with $\vec{C}$.
- $\text{CHECKSSET}(\vec{C}, X)$: returns "yes" iff $\text{CHECK}(\vec{C}, x)$ for some $x \in X$. This checks whether an attested surface set (namely X) is consistent with $\vec{C}$.

These problems place a lower bound on the difficulty of OT generation, since an algorithm that found a reasonable representation of $\text{OPT}(\vec{C}, u)$ (e.g., a DFA) could solve them immediately, and an algorithm that found an exemplar $x \in \text{OPT}(\vec{C}, u)$ could solve all but CHECKSSET immediately. §7 will relate them to OT learning.

6.1 Past Results

Under finite-state assumptions, Ellison (1994) showed that for any fixed $\vec{C}$, a representation of $\text{OPT}(\vec{C}, u)$ could be generated in time $O(|u| \log |u|)$, making all the above problems tractable. However, Eisner (1997a) showed generation to be intractable when $\vec{C}$ was not fixed, but rather considered to be part of the input—as is the case in an algorithm like RCDALL that learns rankings. Specifically, Eisner showed that OPTVALZ is NP-hard. Similarly, Wareham (1998, theorem 4.6.4) showed that a version of

[8] We expect attested sets X_i to be very large—especially in the more general case where they reflect uncertainty about the underlying form. That is why we describe them compactly by DFAs. A universal constraint set $\mathcal{C}$ would also have to be very large (footnote 4).

[9] Allowing directional constraints would not change any of the classifications in this paper.

[10] All these functions take an additional argument Gen, which we suppress for readability.

BEATABLE is NP-hard.[11] (We will obtain more precise classifications below.)

To put this another way, the worst-case complexity of generation problems is something like $O(|u| \log |u|)$ times a term exponential in $|\vec{C}|$.

Thus there are *some* grammars for which generation is very difficult by any algorithm. So when testing exponentially many rankings (§5), a learner may need to spend exponential time testing an individual ranking.

We offer an intuition as to why generation can be so hard. In successive-winnowing algorithms like that of (Eisner, 1997a), the candidate set begins as a large simple set such as Σ^*, and is filtered through successive constraints to end up (typically) as a small simple set such as the singleton $\{x_1\}$. Both these sets can be represented and manipulated as small DFAs. The trouble is that intermediate candidate sets may be complex and require exponentially large DFAs to represent. (Recall that the intersection of DFAs can grow as the product of their sizes.)

For example, Eisner's (1997a) NP-hardness construction (see §6.1) led to such an intermediate candidate set, consisting of all permutations of n digits. Such a set arises simply from a hierarchy such as $\langle$PROJECT$_1, \ldots$ PROJECT$_n$, SHORT$\rangle$, where PROJECT$_j(x) = 0$ provided that j appears (at least once) in x, and SHORT$(x) = |x|$. (Adding a lower-ranked constraint that prefers x to encode a path in a graph G forces OPT to search for a Hamilton path in G, which demonstrates NP-hardness of OPTVALZ.)

6.2 Relevant Complexity Classes

The reader may recall that $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP} \subseteq \mathsf{NP} \cup \mathsf{coNP} \subseteq D^p \subseteq \Delta_2^p = \mathsf{P}^{\mathsf{NP}} \subseteq \Sigma_2^p = \mathsf{NP}^{\mathsf{NP}}$. We will review these classes as they arise. They are classes of **decision problems**, i.e., functions taking values in {yes,no}. Hardness and completness for such classes are defined via many-one (Karp) reductions: g is at least as hard as f iff $(\forall x) f(x) = g(T(x))$ for some function $T(x)$ computable in polynomial time.

OptP is a class of *integer*-valued functions, introduced and discussed by Krentel (1988). Recall that NP is the class of decision problems that can be solved in polytime by a nondeterministic Turing machine (NDTM): each control branch of the machine checks a different possibility and gives a yes/no answer, and the machine returns the *disjunction* of the answers. For coNP, the machine returns the *conjunction* of the answers. For OptP, each branch writes a binary number, and the machine returns the *minimum* (or maximum) of these answers.

A canonical example (analogous to OPTVAL) is the Traveling Salesperson problem—finding the minimum cost TSPVAL(G) of all tours of an integer-weighted graph G. It is OptP-complete in the sense that all functions f in OptP can be **metrically reduced** to it (Krentel, 1988, p. 493). A metric reduction solves an instance of f by transforming it to an instance of g and then appropriately transforming the integer result of g: $(\forall x) f(x) = T_2(x, g(T_1(x)))$ for some polytime-computable functions $T_1 : \Sigma^* \to \Sigma^*$ and $T_2 : \Sigma^* \times \mathbf{N} \to \mathbf{N}$.

Krentel showed that OptP-complete problems yield complete problems for other classes under broad conditions. The question TSPVAL$(G) \leq k$ is of course the classical TSP decision problem, which is NP-complete. (It is analogous to BEATABLE.) The reverse question TSPVAL$(G) \geq k$ (which is related to CHECK) is coNP-complete. The question TSPVAL$(G) = k$ (analogous to BEST) is therefore in the class $D^p = \{L_1 \cap L_2 : L_1 \in \mathsf{NP} \text{ and } L_2 \in \mathsf{coNP}\}$ (Papadimitriou and Yannakakis, 1982), and it is complete for that class. Finally, suppose we wish to ask whether the optimal tour is unique (like OPTVALZ and CHECKSSET, this asks about a complex property of the optimum). Papadimitriou (1984) first showed this question to be complete for $\Delta_2^p = \mathsf{P}^{\mathsf{NP}}$, the class of languages decidable in polytime by deterministic Turing machines that have unlimited access to an oracle that can answer NP questions in unit time. (Such a machine can certainly *decide* uniqueness: It can compute the integer TSPVAL(G) by binary search, asking the oracle for various k whether or not TSPVAL$(G) \leq k$, and then ask it a final NP question: do there exist two distinct tours with cost TSPVAL(G)?)

6.3 New Complexity Results

It is quite easy to show analogous results for OT generation. Our main tool will be one of Krentel's (1988) OptP-complete problems: Minimum Satisfying Assignment. If ϕ is a CNF

[11] Wareham also gave hardness results for versions of BEATABLE where some parameters are bounded or fixed.

boolean formula on n variables, then $\textsc{Msa}(\phi)$ returns the lexicographically minimal bitstring $b_1 b_2 \cdots b_n$ that represents a satisfying assignment for ϕ, or 1^n if no such bitstring exists.[12]

We consider only problems where we can compute $C_j(x)$, or determine whether $x \in \mathsf{Gen}(u)$, in polytime. We further assume that Gen produces only candidates of length polynomial in the size of the problem input—or more weakly, that our functions need not produce correct answers unless at least one *optimal* candidate is so bounded.

Our hardness results (except as noted) apply even to OT grammars with the finite-state and bounded-violations assumptions (§2). In fact, we will assume without further loss of generality (Ellison, 1994; Frank and Satta, 1998; Karttunen, 1998) that constraints are $\{0, 1\}$-valued.

Notation: We may assume that all formulas ϕ use variables from the set $\{v_1, v_2, \ldots\}$. Let $\ell(\phi)$ be the maximum i such that v_i appears in ϕ. We define the constraint C_ϕ to map strings of at least $\ell(\phi)$ bits to $\{0, 1\}$, defining $C_\phi(b_1 \cdots b_n) = 0$ iff ϕ is true when the variables v_i are instantiated to the respective values b_i. So C_ϕ prefers bitstrings that satisfy ϕ.

If we do not make the finite-state assumptions, then any C_ϕ can be represented trivially in size $|\phi|$. Under the finite-state assumptions, however, we must represent C_ϕ as a WDFA. While this is always possible ($\wedge, \vee, \neg$ correspond to intersection, union, and complementation of regular sets), we necessarily take care in this case to use only C_ϕ whose WDFAs are polynomial in $|\phi|$. In particular, if ϕ is a disjunction of (possibly negated) literals, such as $b_2 \vee b_3 \vee \neg b_7$, then the WDFA needs only $\ell(\phi) + 2$ states.

We begin by showing that $\textsc{OptVal}(\vec{C}, u)$ is OptP-complete. It is obvious under our restrictions that it is in the class OptP—indeed it is a perfect example. Each nondeterministic branch of the machine considers some string x of length $\leq p(|u|)$, simply writing the bitstring $\vec{C}(x)$ if $x \in \mathsf{Gen}(u)$ and 1^n otherwise.

To show OptP-hardness, we metrically reduce $\textsc{Msa}(\phi)$ to $\textsc{OptVal}$, where $\phi = \bigwedge_{i=1}^m D_i$ is in CNF. Let $n = \ell(\phi)$, and put $\mathcal{L} = \{\epsilon\}$ and $\mathsf{Gen}(\epsilon) = \{0, 1\}^n$. Then $\textsc{Msa}(\phi) =$ the last n bits of $\min(0^m 1^n, \textsc{OptVal}(\langle C_{D_1}, \ldots C_{D_m}, C_{\neg v_1}, \ldots C_{\neg v_n}\rangle, \epsilon))$.[13]

Because $\textsc{OptVal}$ is OptP-complete, Krentel's theorem 3.1 says it is complete for $\mathsf{FP}^{\mathsf{NP}}$, the set of functions computable in polynomial time using an oracle for NP. This is the function class corresponding to the decision class $\mathsf{P}^{\mathsf{NP}} = \Delta_2^p$.

Next we show that $\textsc{Beatable}(\vec{C}, u, \vec{k})$ is NP-complete. It is obviously in NP. For NP-hardness, observe that $\text{CNF-}\textsc{Sat}(\phi) = \textsc{Beatable}(\langle C_{D_1}, \ldots C_{D_m}\rangle, \epsilon, \langle 0, 0, \ldots 0, 1\rangle)$, where again $\phi = \bigwedge_{i=1}^m D_i$, $n = \ell(\phi)$, and $\mathsf{Gen}(\epsilon) = \{0, 1\}^n$.

Next consider $\textsc{Check}(\vec{C}, x)$. This is simply $\neg\textsc{Beatable}(\vec{C}, u_x, \vec{C}(x))$. Even when restricted to calls of this form, $\textsc{Beatable}$ remains NP-complete. To show this, we tweak the above construction so we can write $\vec{C}(x)$ (for some x) in place of $\langle 0, 0, \ldots 0, 1\rangle$. Add the new element ϵ to $\mathsf{Gen}(\epsilon)$, and extend the constraint definitions by putting $C_{D_i}(\epsilon) = 0$ iff $i < m$. Then $\text{CNF-}\textsc{Sat}(\phi) = \textsc{Beatable}(\vec{C}, \epsilon, \vec{C}(\epsilon))$. Therefore $\textsc{Check}$ is coNP-complete.

Next we consider $\textsc{Best}(\vec{C}, u, \vec{k})$. This problem is in D^p for the same simple reason that the question $\textsc{TspVal}(G) = k$ is (see above). If we do not make the finite-state assumptions, it is also D^p-hard by reduction from the D^p-complete language $\textsc{Sat-Unsat} = \{\phi \# \psi : \phi \in \textsc{Sat}, \psi \notin \textsc{Sat}\}$ (Papadimitriou and Yannakakis, 1982), as follows: $\textsc{Sat-Unsat}(\phi \# \psi) = \textsc{Best}(\langle C_\phi, C_\psi\rangle, \epsilon, \langle 0, 1\rangle)$, renaming variables as necessary so that ϕ uses only $v_1, \ldots v_r$ and ψ uses only $v_{r+1}, \ldots v_s$, and $\mathsf{Gen}(\epsilon) = \{0, 1\}^{r+s}$.

It is not clear whether $\textsc{Best}$ remains D^p-hard under the finite-state assumptions. But consider a more flexible variant $\textsc{Range}(\vec{C}, u, \vec{k_1}, \vec{k_2})$ that asks whether $\textsc{OptVal}(\vec{C}, u)$ is between $\vec{k_1}$ and $\vec{k_2}$ inclusive. This is also in D^p, and is D^p-hard because $\textsc{Sat-Unsat}(\phi \# \psi) = \textsc{Range}(\langle C_{D_1}, \ldots C_{D_m}, C_{D'_1}, \ldots C_{D'_{m'}}\rangle, \epsilon, \langle 0, \ldots 0, 0, \ldots 1\rangle, \langle 0, \ldots 0, 1, \ldots 1\rangle$, where ϕ, ψ, Gen are as before and $\phi = \bigwedge_{i=1}^m D_i$, $\psi = \bigwedge_{i=1}^{m'} D'_i$.

Finally, we show that the decision prob-

[12] Krentel's presentation is actually in terms of Maximum Satisfying Assignment, which merely reverses the roles of 0 and 1. Also, Krentel does not mention that ϕ can be restricted to (3)CNF, but his proof of OptP-hardness makes this important fact clear.

[13] Without the finite-state assumptions, we could more simply write $\textsc{Msa}(\phi) = \textsc{OptVal}(\langle C_{\psi_1}, \ldots C_{\psi_n}\rangle, \epsilon)$, where $\psi_j = \phi \wedge \neg v_j$.

lem $\textsc{CheckSSet}(\vec{C}, X)$ is Δ_2^p-complete. It is in Δ_2^p by an algorithm similar to the one used for TSP uniqueness above: since BEATABLE can be determined by an NP oracle, we can find $\textsc{OptVal}(\vec{C}, u)$ by binary search.[14] An additional call to an NP oracle decides $\textsc{CheckSSet}(\vec{C}, X)$ by asking whether there is any $x \in X$ such that $\vec{C}(x) = \textsc{OptVal}(\vec{C}, u)$.

The reduction to show Δ_2^p-hardness is from a Δ_2^p-complete problem exhibited by Krentel (1988, theorem 3.4): $\textsc{Msa}_{lsb}$ accepts ϕ iff the final (least significant) bit of $\textsc{Msa}(\phi)$ is 0. Given ϕ, we use the same grammar as when we reduced MSA to OPTVAL. $\textsc{Msa}_{lsb}$ accepts iff OPTVAL found a satisfying assignment *and* the last bit of this optimal assignment was 0: i.e., $\textsc{Msa}_{lsb}(\phi) = \textsc{CheckSSet}(\langle C_{D_1}, \ldots C_{D_m}, C_{\neg v_1}, \ldots C_{\neg v_n}\rangle, 0^m\{0,1\}^{n-1}0)$.[15]

Note that we did not have to use an unreasonable attested surface set as in §5.1. The set $0^m\{0,1\}^{n-1}0$ means that the learner has observed only certain bits of the utterance—exactly the kind of partial observation that we expect. So even some restriction to "reasonable" attested sets is unlikely to help.

7 Complexity of OT Ranking

We now consider two ranking problems. These ask whether $\mathcal{C}$ can be ranked in a manner consistent with attested forms or attested sets:

- $\textsc{Rankable}(\mathcal{C}, \{x_1, \ldots x_m\})$: returns "yes" iff there is a ranking $\vec{C}$ of $\mathcal{C}$ such that $\textsc{Check}(\vec{C}, x_i)$ for all i.
- $\textsc{RankableSSet}(\mathcal{C}, \{X_i, \ldots X_m\})$: returns "yes" iff there is a ranking $\vec{C}$ of $\mathcal{C}$ such that $\textsc{CheckSSet}(\vec{C}, X_i)$ for all i.

We do not have an exact classification of RANKABLE at this time. Interestingly, the special case where $m = 1$ and the constraints take values in $\{0, 1\}$ (which has sufficed to show most of our hardness results) is coNP-complete—the same as CHECK, which only verifies a solution.

[14] This takes polynomially many steps provided that the integer $C_i(x)$ is bounded by $2^{q(|x|)}$ for some polynomial q (as it is under the finite-state assumptions). We have already assumed above that $|x|$ itself is polynomial on the input size, at least for optimal x.

[15] We can similarly show that OPTVALZ is not merely NP-hard (Eisner, 1997a) but Δ_2^p-complete, at least if we drop the finite-state assumptions: $\textsc{Msa}_{lsb}(\phi) = \textsc{OptValZ}(\langle C_\phi, C_{\neg v_1}, \ldots C_{\neg v_{n-1}}, C_{\phi \wedge \neg v_n}\rangle, \epsilon)$.

Here RANKABLE need only ask whether there exists any $y \in \mathsf{Gen}(u_{x_1})$ that satisfies a proper superset of the constraints that x_1 satisfies. For if so, x_1 cannot be optimal under any ranking, and if not, then we can simply rank the constraints that x_1 satisfies above the others. This immediately implies that the special case is in coNP. It also implies it is coNP-hard: using the grammar from our proof that CHECK is coNP-hard (§6.3), we write $\text{CNF-}\textsc{Sat}(\phi) = \neg\textsc{Rankable}(\mathcal{C}, \{\epsilon\})$.

As an upper bound on the complexity of RANKABLE, we saw in §4.1 that the RCDALL algorithm of §4 can decide RANKABLE with $O(n^2 m)$ calls to OPT (where $n = |\mathcal{C}|$). In fact, it suffices to call CHECK rather than OPT (since RCDALL only tests whether $x_i \in \textsc{Opt}(\cdots)$). Since $\textsc{Check} \in \mathsf{coNP}$, it follows that RANKABLE is in $\mathsf{P}^{\mathsf{coNP}} = \mathsf{P}^{\mathsf{NP}} = \Delta_2^p$.

Notice that while Tesar's EDCD and MRCD algorithms (§4.2) can also decide RANKABLE with polynomially many calls to OPT—or, better, to OPTVAL, since they do not use y except to compute $\vec{C}(y)$. But they cannot get by with calls to CHECK as RCDALL does. OPTVAL is "harder" than CHECK ($\mathsf{FP}^{\mathsf{NP}}$-complete vs. coNP-complete). This is a reason to prefer RCDALL to EDCD and MRCD.

RANKABLESSET is certainly in Σ_2^p, since it may be phrased in $\exists\forall$ form as $(\exists\vec{C}, \{x_i \in X_i\})$ $(\forall i, y_i \in \mathsf{Gen}(u_{x_i}))\ \vec{C}(x_i) \leq \vec{C}(y_i)$. We saw in §5 that it is NP-hard even when the constraints interact simply. One suspects it is Δ_2^p-hard, since merely verifying a solution (i.e., CHECKSSET) is Δ_2^p-complete (§6.3). We now show that is actually Σ_2^p-hard and therefore Σ_2^p-complete.

The proof is by reduction from the canonical Σ_2^p-complete problem $\textsc{QSat}_2(\phi, r)$, where $\phi = \bigwedge_{i=1}^m D_i$ is a CNF formula with $\ell(\phi) \geq r \geq 0$. This returns "yes" iff

$$\exists b_1, \ldots b_r \neg\exists b_{r+1}, \ldots b_s \phi(b_1, \ldots b_s),$$

where $s \overset{\text{def}}{=} \ell(\phi)$ and $\phi(b_1, \ldots b_s)$ denotes the truth value of ϕ when the variables $v_1, \ldots v_s$ are bound to the respective binary values $b_1 \ldots b_s$.

Given an instance of $\textsc{QSat}_2$ as above, put $\mathcal{L} = \{\epsilon\}$ and $\mathsf{Gen}(\epsilon) = \{0,1\}^{r+s} \cup X$ where $X = \{0,1\}^r 2$. Let $\mathcal{C} = \{C_{D_1}, \ldots C_{D_m}, C_{v_1}, \ldots C_{v_r}, C_{\neg v_1}, \ldots C_{\neg v_r}, *X\}$, where all constraints have range $\{0, 1\}$, we extend C_{D_i} over X by defining it to be satisfied (i.e., take value

0) on all candidates in X, and we define $*X$ to be satisfied on exactly those candidates not in X. As before, C_{v_i} and $C_{\neg v_i}$ are satisfied on a candidate iff its ith bit is 1 or 0 respectively, regardless of whether the candidate is in X.

We now claim that $\text{QSAT}_2(\phi, r) = \text{RANKABLESSET}(\mathcal{C}, \{X\})$. The following terminology will be useful in proving this: Given a bit sequence $\vec{b} = b_1, \ldots b_r$, define a **$\vec{b}$-satisfier** to be a bit string $b_1 \cdots b_r b_{r+1} \ldots b_s$ such that $\phi(b_1, \ldots b_s)$. For $1 \le i \le r$, let $B_i, \bar{B}_i$ denote the constraints $C_{v_i}, C_{\neg v_i}$ respectively if $b_i = 1$, or vice-versa if $b_i = 0$. We then say that a ranking $\vec{C}$ of $\mathcal{C}$ is **$\vec{b}$-compatible** if B_i precedes $\bar{B}_i$ in $\vec{C}$ for every $1 \le i \le r$.

Observe that a candidate $y \in \mathsf{Gen}(\epsilon)$ is a $\vec{b}$-satisfier iff it satisfies the constraints $B_1, \ldots B_r$ and $C_{D_1}, \ldots C_{D_m}$ and $*X$. From this it is not difficult to see that if $\vec{C}$ is a $\vec{b}$-compatible ranking, then y beats x (i.e., $\vec{C}(y) < \vec{C}(x)$) for any $\vec{b}$-satisfier y and any $x \in X$.[16]

Suppose $\text{RANKABLESSET}(\mathcal{C}, \{X\})$. Then choose $x \in X$ and $\vec{C}$ a ranking of $\mathcal{C}$ such that x is optimal (i.e., $\text{CHECK}(\vec{C}, x)$). For each $1 \le i \le r$, let $b_i = 1$ if C_{v_i} is ranked before $C_{\neg v_i}$ in $\vec{C}$, otherwise $b_i = 0$. Then $\vec{C}$ is a $\vec{b}$-compatible ranking. Since $x \in X$ is optimal, there must be no $\vec{b}$-satisfiers y, i.e., $\text{QSAT}_2(\phi, r)$.

Conversely, suppose $\text{QSAT}_2(\phi, r)$. This means we can choose $b_1, \ldots b_r$ such that there are no $\vec{b}$-satisfiers. Let $\vec{C} = \langle C_{D_1}, \ldots C_{D_m}, B_1, \ldots B_r, \bar{B}_1, \ldots \bar{B}_r, *X \rangle$. Observe that $x = b_1 \cdots b_r 2 \in X$ satisfies the first $m + r$ of the constraints; this is optimal (i.e., $\text{CHECK}(\vec{C}, x)$), since any better candidate would have to be a $\vec{b}$-satisfier.[17] Hence there is a ranking $\vec{C}$ consistent with X, i.e., $\text{RANKABLESSET}(\mathcal{C}, \{X\})$.

8 Optimization vs. Derivation

The above results mean that OT generation and ranking are hard. We will now see that they are harder than the corresponding problems in deterministic derivational theories, assuming that the complexity classes discussed are distinct.

[16] $\vec{C}(y) = \vec{C}(x)$ is impossible: only x violates $*X$. And $\vec{C}(y) > \vec{C}(x)$ is impossible, for if x satisfies any constraint that y violates, namely some $\bar{B}_i$, then it violates a higher-ranked constraint that y satisfies, namely B_i.

[17] Since it would have to satisfy the first $m + r$ constraints plus a later constraint, which could only be $*X$.

A **derivational grammar** consists of the following elements (cf. §2):

- an alphabet Σ;
- a set $\mathcal{L} \subseteq \Sigma^*$ of underlying forms;
- a vector $\vec{R} = \langle R_1, \ldots R_n \rangle$ of **rules**, each of which is a function from Σ^* to Σ^*.

The grammar maps each $x \in \mathcal{L}$ to $\vec{R}(x) \stackrel{\text{def}}{=} R_n \circ \cdots \circ R_2 \circ R_1(x)$. If all the rules are polytime-computable (i.e., in the function class FP), then so is $\vec{R}$. (By contrast, the OT analogue OPT is complete for the function class $\mathsf{FP}^{\mathsf{NP}}$.) It follows that the derivational analogues of the decision problems given at the start of §6 are in P[18] (whereas we have seen that the OT versions range from NP-complete to Δ_2^p-complete).

How about learning? The **rule ordering problem** ORDERABLESSET takes as input a set $\mathcal{R}$ of possible rules, a unary integer n, and a set of pairs $\{(u_1, X_1), \ldots (u_m, X_m)\}$ where $u_i \in \Sigma^*$ and $X_i \subseteq \Sigma^*$. It returns "yes" iff there is a a rule sequence $\vec{R} \in \mathcal{R}^n$ such that $(\forall i)\vec{R}(u_i) \in X_i$. It is clear that this problem is in NP. This makes it easier than its OT analogue RANKABLESSET and possibly easier than RANKABLE.

For interest, we show that ORDERABLESSET and its restricted version ORDERABLE (where the attested sets X_i are replaced by attested forms x_i) are NP-complete. As usual, our result holds even with finite-state restrictions: we require the rules in $\mathcal{R}$ to be regular relations (Johnson, 1972). The hardness proof is by reduction from Hamilton Path (defined in §5.1). Given a directed graph G with vertices $1, 2, \ldots n$, put $\Sigma = \{\#, 0, 1, 2, \ldots n\}$. Each string we consider will be either ϵ or a permutation of Σ. Define MOVE_j to be a rule that maps $\alpha j \beta \# \gamma i$ to $\alpha \beta \# \gamma i j$ for any $i, j \in \Sigma, \alpha, \beta, \gamma \in \Sigma^*$ such that $i = 0$ or else G has an edge from i to j, and acts as the identity function on other strings. Also define ACCEPT to be a rule that maps $\#\alpha$ to ϵ for any $\alpha \in \Sigma^*$, and acts as the identity function on other strings. Now $\text{ORDERABLE}(\{\text{MOVE}_1, \ldots \text{MOVE}_n, \text{ACCEPT}\}, n+1, \{(12 \cdots n\#0, \epsilon)\})$ decides whether G has a Hamilton path.

[18] However, Wareham (1998) analyzes a more powerful derivational approach where the rules are nondeterministic: each R_i is a relation rather than a function. Wareham shows that generation in this case is NP-hard (Theorem 4.3.3.1). He does not consider learning.

9 Conclusions

The reader is encouraged to see the abstract for a summary of our most important results. Our main conclusion is a warning that OT may carry huge computational burdens. When formulating the OT learning problem, even small nods in the direction of realism quickly drive the complexity from linear-time up through coNP (for multiple competitors) into the higher complexity classes (for multiple possible surface forms).

Intuitively, an OT learner must both pick a constraint ranking (∃) and check that an attested form beats all competitors under that ranking (∀). By contrast, a derivational learner need only pick a rule ordering (∃).

One constraint ranking problem we consider, RankableSSet, is in fact a rare "natural" example of a problem that is complete for the higher complexity class Σ_2^p (∃∀). Some other learning problems were already known to be Σ_2^p-complete (Ko and Tzeng, 1991), but ours is different in that it uses only positive evidence.

This paper leaves some theoretical questions open. Most important is the exact classification of Rankable. Second, we are interested in any cases where problem variants (e.g., accepting vs. rejecting the finite-state assumptions) differ in complexity. Third, in the same spirit, parameterized complexity analyses (Wareham, 1998) may help identify sources of hardness.

We are also interested in more realistic versions of the phonology learning problem. We are especially interested in the possibility that $\mathcal{C}$ has internal structure, as discussed in footnote 4, and in the problem of learning from general attested sets, not just attested surface sets.

Finally, in light of our demonstrations that efficient algorithms are highly unlikely for the problems we have considered, we ask: Are there restrictions, reformulations, or randomized or approximate methods that could provably make OT learning tractable in some sense?

References

Paul Boersma. 1997. How we learn variation, optionality, and probability. In *Proc. of the Institute of Phonetic Sciences* 21, U. of Amsterdam, 43–58.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. 1990. *Introduction to Algorithms*. MIT Press.

Jason Eisner. 1997a. Efficient generation in primitive Optimality Theory. In *Proceedings of ACL/EACL*, 313–320, Madrid, July.

Jason Eisner. 1997b. What constraints should OT allow? Talk handout, Linguistic Society of America. Rutgers Optimality Archive ROA-204.

Jason Eisner. 2000. Directional constraint evaluation in Optimality Theory. In *Proceedings of COLING*, Saarbrücken, Germany, August.

T. Mark Ellison. 1994. Phonological derivation in Optimality Theory. *Proceedings of COLING*.

Robert Frank and Giorgio Satta. 1998. Optimality Theory and the generative complexity of constraint violability. *Computational Linguistics*, 24(2):307–315.

E. M. Gold. 1967. Language identification in the limit. *Information and Control*, 10:447–474.

C. Douglas Johnson. 1972. *Formal Aspects of Phonological Description*. Mouton.

Mark Johnson. 2000. Context-sensitivity and stochastic "unification-based" grammars. Talk presented at the CLSP Seminar Series, Johns Hopkins University, February.

René Kager. 1999. *Optimality Theory*. Cambridge University Press.

Lauri Karttunen. 1998. The proper treatment of optimality in computational phonology. In *Proceedings of International Workshop on Finite-State Methods in NLP*, 1–12, Bilkent University.

Ker-I Ko and Wen-Guey Tzeng. 1991. Three Σ_2^p-complete problems in computational learning theory. *Computational Complexity*, 1:269–310.

Mark W. Krentel. 1988. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509.

C. H. Papadimitriou and M. Yannakakis. 1982. The complexity of facets (and some facets of complexity). In *Proc. of the 14th Annual Symposium on Theory of Computing*, 255–260, New York. ACM.

Christos H. Papadimitriou. 1984. On the complexity of unique solutions. *JACM*, 31(2):392–400.

A. Prince and P. Smolensky. 1993. Optimality Theory: Constraint interaction in generative grammar. Ms., Rutgers U. and U. Colorado (Boulder).

Bruce Tesar and Paul Smolensky. 1996. Learnability in Optimality Theory (long version). Technical Report JHU-CogSci-96-3, Johns Hopkins University, October. Shortened version appears in *Linguistic Inquiry* **29**:229–268, 1998.

Bruce Tesar and Paul Smolensky. 2000. *Learnability in Optimality Theory*. MIT Press, Cambridge.

Bruce Tesar. 1996. Computing optimal descriptions for Optimality Theory grammars with context-free position structures. In *Proceedings of ACL*.

Bruce Tesar. 1997. Multi-recursive constraint demotion. Rutgers Optimality Archive ROA-197.

Harold Todd Wareham. 1998. *Systematic Parameterized Complexity Analysis in Computational Phonology*. Ph.D. thesis, University of Victoria.

Approximation and Exactness in Finite State Optimality Theory

Dale Gerdemann
University of Tübingen
dg@sfs.nphil.uni-tuebingen.de

Gertjan van Noord
University of Groningen
vannoord@let.rug.nl

Abstract

Previous work (Frank and Satta, 1998; Karttunen, 1998) has shown that Optimality Theory with *gradient* constraints generally is not finite state. A new finite-state treatment of gradient constraints is presented which improves upon the approximation of Karttunen (1998). The method turns out to be exact, and very compact, for the syllabification analysis of Prince and Smolensky (1993).

1 Introduction

Finite state methods have proven quite successful for encoding rule-based generative phonology (Johnson, 1972; Kaplan and Kay, 1994). Recently, however, Optimality Theory (Prince and Smolensky, 1993) has emphasized phonological accounts with default constraints on surface forms. While Optimality Theory (OT) has been successful in explaining certain phonological phenomena such as *conspiracies* (Kisseberth, 1970), it has been less successful for computation. The negative result of Frank and Satta (1998) has shown that in the general case the method of counting constraint violations takes OT beyond the power of regular relations. To handle such constraints, Karttunen (1998) has proposed a finite-state approximation that counts constraint violations up to a predetermined bound. Unlike previous approaches (Ellison, 1994; Walther, 1996), Karttunen's approach is encoded entirely in the finite state calculus, with no extra-logical procedures for counting constraint violations.

In this paper, we will present a new approach that seeks to minimize constraint violations without counting. Rather than counting, our approach employs a filter based on matching constraint violations against violations in alternatively derivable strings. As in Karttunen's counting approach, our approach uses purely finite state methods without extra-logical procedures. We show that our *matching* approach is superior to the *counting* approach for both size of resulting automata and closeness of approximation. The matching approach can in fact exactly model many OT analyses where the counting approach yields only an approximation; yet, the size of the resulting automaton is typically much smaller.

In this paper we will illustrate the matching approach and compare it with the counting approach on the basis of the Prince & Smolensky syllable structure example (Prince and Smolensky, 1993; Ellison, 1994; Tesar, 1995), for each of the different constraint orderings identified in Prince & Smolensky.

2 Finite State Phonology

2.1 Finite State Calculus

Finite state approaches have proven to be very successful for efficient encoding of phonological rules. In particular, the work of Kaplan and Kay (1994) has provided a compiler from classical generative phonology rewriting rules to finite state transducers. This work has clearly shown how apparently procedural rules can be recast in a declarative, reversible framework.

In the process of developing their rule compiler, Kaplan & Kay also developed a high-level finite state calculus. They argue convincingly that this calculus provides an appropriate high-level approach for expressing regular languages and relations. The alternative conception in term of states and transitions can become unwieldy for all but the simplest cases.[1]

[1] Although in some cases such a direct implementation can be much more efficient (Mohri and Sproat, 1996; van Noord and Gerdemann, 1999).

[]	empty string
[E1,E2,...,En]	concatenation of E1...En
{}	empty language
{E1,E2,...,En}	union of E1...En
(E)	grouping for op. precedence
E*	Kleene closure
E+	Kleene plus
E^	optionality
E1 - E2	difference
~E	complement
$ E	containment
E1 & E2	intersection
?	any symbol
E1 x E2	cross-product
A o B	composition
domain(E)	domain of a transduction
range(E)	range of a transduction
identity(E)	identity transduction[2]
inverse(E)	inverse transduction

Table 1: Regular expression operators.

Kaplan & Kay's finite state calculus now exists in multiple implementations, the most well-known of which is that of Karttunen et al. (1996). In this paper, however, we will use the alternative implementation provided by the FSA Utilities (van Noord, 1997; van Noord, 1999; van Noord and Gerdemann, 1999). The FSA Utilities allows the programmer to introduce new regular expression operators of arbitrary complexity. This higher-level interface allows us to express our algorithm more easily. The syntax of the FSA Utilities calculus is summarized in Table 1.

The finite state calculus has proven to be a very useful tool for the development of higher-level finite state operators (Karttunen, 1995; Kempe and Karttunen, 1996; Karttunen, 1996; Gerdemann and van Noord, 1999). An interesting feature of most such operators is that they are implemented using a generate-and-test paradigm. Karttunen (1996), for example, introduces an algorithm for a leftmost-longest replacement operator. Somewhat simplified, we may view this algorithm as having two steps. First, the generator freely marks up possible replacement sites. Then the tester, which is an identity transducer, filters out those cases not conforming to the leftmost-longest strategy. Since the generator and tester are both implemented as transducers, they can be composed into a single transducer, which eliminates the inefficiency normally associated with generate-and-test algorithms.

[2]If an expression for a recognizer occurs in a context where a transducer is required, the identity operation will be used implicitly for coercion.

2.2 Finite State Optimality Theory

The generate-and-test paradigm initially appears to be appropriate for optimality theory. If, as claimed in Ellison (1994), *Gen* is a regular relation and if each constraint can be implemented as an identity transducer, then optimality theory analyses could be implemented as in fig. 1.

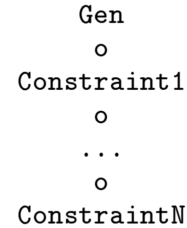


Figure 1: Optimality Theory as Generate and Test

The problem with this simple approach is that in OT, a constraint is allowed to be violated if none of the candidates satisfy that constraint. Karttunen (1998) treats this problem by providing a new operator for *lenient* composition, which is defined in terms of the auxiliary operation of priority union. In the FSA Utilities calculus, these operations can be defined as:[3]

```
macro(priority_union(Q,R),
      {Q, ~domain(Q) o R}).
macro(lenient_composition(S,C),
      priority_union(S o C, S)).
```

The effect here is that the lenient composition of S and C is the composition of S and C, except for those elements in the domain of S that are not mapped to anything by S o C. For these elements not in the domain of S o C, the effect is the same as the effect of S alone. We use the

[3]The notation macro(Expr1,Expr2) is used to indicate that the regular expression Expr1 is an abbreviation for the expression Expr2. Because Prolog variables are allowed in both expressions this turns out to be an intuitive and powerful notation (van Noord and Gerdemann, 1999).

notation `S lc C` as a succinct notation for the lenient composition of S and C. Using lenient composition an OT analysis can be written as in fig. 2.

```
      Gen
       lc
  Constraint1
       lc
       ...
       lc
  ConstraintN
```

Figure 2: Optimality Theory as Generate and Test with Lenient Composition

The use of lenient composition, however, is not sufficient for implementing optimality theory. In general, a candidate string can violate a constraint multiple times and candidates that violate the constraint the least number of times need to be preferred. Lenient composition is sufficient to prefer a candidate that violates the constraint 0 times over a candidate that violates the constraint at least once. However, lenient composition cannot distinguish two candidates if the first contains one violation, and the second contains at least two violations.

The problem of implementing optimality theory becomes considerably harder when constraint violations need to be counted. As Frank and Satta (1998) have shown, an OT describes a regular relation under the assumptions that `Gen` is a regular relation, and each of the constraints is a regular relation which maps a candidate string to a natural number (indicating the number of constraint violations in that candidate), where the range of each constraint is finite. If constraints are defined in such a way that there is no bound to the number of constraint violations that can occur in a given string, then the resulting OT may describe a relation that is not regular. A simple example of such an OT (attributed to Markus Hiller) is the OT in which the inputs of interest are of the form `[a*,b*]`, *Gen* is defined as a transducer which maps all `a`'s to `b`'s and all `b`'s to `a`'s, or alternatively, it performs the identity map on each `a` and `b`:

```
{[(a x b)*,(b x a)*],
 [(a x a)*,(b x b)*]}
```

This OT contains only a single constraint, $^*A$: a string should not contain `a`. As can easily be verified, this OT defines the relation $\{(a^n b^m, a^n b^m)|n \leq m\} \cup \{(a^n b^m, b^n a^m)|m \leq n\}$, which can easily be shown to be non-regular.

Although the OT described above is highly unrealistic for natural language, one might nevertheless expect that a constraint on syllable structure in the analysis of Prince & Smolensky would require an unbounded amount of counting (since words are of unbounded length), and that therefore such analyses would not be describable as regular relations. An important conclusion of this paper is that, contrary to this potential expectation, such cases in fact *can* be shown to be regular.

2.3 Syllabification in Finite State OT

In order to illustrate our approach, we will start with a finite state implementation of the syllabification analysis as presented in chapter 6 of Prince and Smolensky (1993). This section is heavily based on Karttunen (1998), which the reader should consult for more explanation and examples.

The inputs to the syllabification OT are sequences of consonants and vowels. The input will be marked up with *onset*, *nucleus*, *coda* and *unparsed* brackets; where a syllable is a sequence of an optional onset, followed by a nucleus, followed by an optional coda. The input will be marked up as a sequence of such syllables, where at arbitrary places *unparsed* material can intervene. The assumption is that an unparsed vowel or consonant is not spelled out phonetically. Onsets, nuclei and codas are also allowed to be empty; the phonetic interpretation of such constituents is *epenthesis*.

First we give a number of simple abbreviations:

```
macro(cons,
      {b,c,d,f,g,h,j,k,l,m,n,
       p,q,r,s,t,v,w,x,y,z} ).
macro(vowel,   {a,e,o,u,i}).

macro(o_br,    'O['). % onset
macro(n_br,    'N['). % nucleus
macro(d_br,    'D['). % coda
macro(x_br,    'X['). % unparsed
macro(r_br,    ']').
macro(bracket,
      {o_br,n_br,d_br,x_br,r_br}).
```

```
macro(onset,    [o_br,cons^  ,r_br]).
macro(nucleus, [n_br,vowel^ ,r_br]).
macro(coda,     [d_br,cons^  ,r_br]).
macro(unparsed,[x_br,letter ,r_br]).
```

Following Karttunen, *Gen* is formalized as in fig. 3. Here, `parse` introduces *onset*, *coda* or *unparsed* brackets around each consonant, and *nucleus* or *unparsed* brackets around each vowel. The `replace(T,Left,Right)` transducer applies transducer `T` obligatory within the contexts specified by `Left` and `Right` (Gerdemann and van Noord, 1999). The `replace(T)` transducer is an abbreviation for `replace(T,[],[])`, i.e. `T` is applied everywhere. The *overparse* transducer introduces optional 'empty' constituents in the input, using the *intro_each_pos* operator.[4]

In the definitions for the constraints, we will deviate somewhat from Karttunen. In his formalization, a constraint simply describes the set of strings which do not violate that constraint. It turns out to be easier for our extension of Karttunen's formalization below, as well as for our alternative approach, if we return to the concept of a constraint as introduced by Prince and Smolensky where a constraint adds *marks* in the candidate string at the position where the string violates the constraint. Here we use the symbol `@` to indicate a constraint violation. After checking each constraint the markers will be removed, so that markers for one constraint will not be confused with markers for the next.

```
macro(mark_violation(parse),
     replace(([] x @),x_br,[]).

macro(mark_violation(no_coda),
     replace(([] x @),d_br,[]).

macro(mark_violation(fill_nuc),
     replace(([] x @),[n_br,r_br],[])).

macro(mark_violation(fill_ons),
     replace(([] x @),[o_br,r_br],[])).

macro(mark_violation(have_ons),
     replace(([] x @),[],n_br)
                      o
     replace((@ x []),onset,[])).
```

The *parse* constraint simply states that a candidate must not contain an unparsed constituent. Thus, we add a mark after each unparsed bracket. The *no_coda* constraint is similar: each coda bracket will be marked. The *fill_nuc* constraint is only slightly more complicated: each sequence of a nucleus bracket immediately followed by a closing bracket is marked. The *fill_ons* constraint treats empty onsets in the same way. Finally, the *have_ons* constraint is somewhat more complex. The constraint requires that each nucleus is preceded by an onset. This is achieved by marking all nuclei first, and then removing those marks where in fact an onset is present.

This completes the building blocks we need for an implementation of Prince and Smolensky's analysis of syllabification. In the following sections, we present two alternative implementations which employ these building blocks. First, we discuss the approach of Karttunen (1998), based on the lenient composition operator. This approach uses a *counting* approach for multiple constraint violations. We will then present an alternative approach in which constraints eliminate candidates using *matching*.

3 The Counting Approach

In the approach of Karttunen (1998), a candidate set is leniently composed with the set of strings which satisfy a given constraint. Since we have defined a constraint as a transducer which marks candidate strings, we need to alter the definitions somewhat, but the resulting transducers are equivalent to the transducers produced by Karttunen (1998). We use the (left-associative) *optimality operator* `oo` for applying an OT constraint to a given set of candidates:[5]

[4] An alternative would be to define *overparse* with a Kleene star in place of the option operator. This would introduce unbounded sequences of empty segments. Even though it can be shown that, with the constraints assumed here, no optimal candidate ever contains two empty segments in a row (proposition 4 of Prince and Smolensky (1993)) it is perhaps interesting to note that defining *Gen* in this alternative way causes cases of infinite ambiguity for the counting approach but is unproblematic for the matching approach.

[5] The operators 'o' and 'lc' are assumed to be left associative and have equal precedence.

```
macro(gen,       {cons,vowel}*
                        o
                    overparse
                        o
                      parse
                        o
                 syllable_structure ).

macro(parse, replace([[] x {o_br,d_br,x_br},cons, [] x r_br])
                                  o
             replace([[] x {n_br,x_br},     vowel,[] x r_br])).

macro(overparse,intro_each_pos([{o_br,d_br,n_br},r_br]^)).

macro(intro_each_pos(E), [[ [] x E, ?]*,[] x E]).

macro(syllable_structure,ignore([onset^,nucleus,coda^],unparsed)*).
```

Figure 3: The definition of *Gen*

```
macro(Cands oo Constraint,
          Cands
            o
  mark_violation(Constraint)
            lc
        ~ ($ @)
            o
     { @ x [], ? - @}*    ).
```

Here, the set of candidates is first composed with the transducer which marks constraint violations. We then leniently compose the resulting transducer with `~($ @)`[6], which encodes the requirement that no such marks should be contained in the string. Finally, the remaining marks (if any) are removed from the set of surviving candidates. Using the optimality operator, we can then combine *Gen* and the various constraints as in the following example (equivalent to figure 14 of Karttunen (1998)):

```
macro(syllabify, gen
                  oo
              have_ons
                  oo
              no_coda
                  oo
              fill_nuc
                  oo
               parse
                  oo
              fill_ons     ).
```

As mentioned above, a candidate string can violate a constraint multiple times and candidates that violate the constraint the least number of times need to be preferred. Lenient composition cannot distinguish two candidates if the first contains one violation, and the second contains at least two violations. For example, the above `syllabify` transducer will assign three outputs to the input `bebop`:

```
O[b]N[e]X[b]X[o]X[p]
O[b]N[e]O[b]N[o]X[p]
X[b]X[e]O[b]N[o]X[p]
```

In this case, the second output should have been preferred over the other two, because the second output violates 'Parse' only once, whereas the other outputs violate 'Parse' three times. Karttunen recognizes this problem and proposes to have a sequence of constraints Parse0, Parse1, Parse2 ... ParseN, where each ParseX constraint requires that candidates not contain more than X unparsed constituents.[7] In this case, the resulting transducer only *approximates*

[6] As explained in footnote 2, this will be coerced into an identity transducer.

[7] This construction is similar to the construction in Frank and Satta (1998), who used a suggestion in Ellison (1994).

the OT analysis, because it turns out that for any X there are candidate strings that this transducer fails to handle correctly (assuming that there is no bound on the length of candidate strings).

Our notation is somewhat different, but equivalent to the notation used by Karttunen. Instead of a sequence of constraints Cons0 ... ConsX, we will write `Cands oo Prec :: Cons`, which is read as: apply constraint `Cons` to the candidate set `Cands` with *precision* `Prec`, where "precision" means the predetermined bound on counting. For example, a variant of the `syllabify` constraint can be defined as:

```
macro(syllabify, gen
                   oo
                 have_ons
                   oo
                 no_coda
                   oo
                 1 :: fill_nuc
                   oo
                 8 :: parse
                   oo
                 fill_ons     ).
```

Using techniques described in §5, this variant can be shown to be *exact* for all strings of length ≤ 10. Note that if no precision is specified, then a precision of 0 is assumed.

This construct can be defined as follows (in the actual implementation the regular expression is computed dynamically based on the value of `Prec`):

```
macro(Cands oo 3 :: Constraint,
          Cands
            o
  mark_violation(Constraint)
            lc
  ~ ([($ @),($ @),($ @),($ @)])
            lc
    ~ ([($ @),($ @),($ @)])
            lc
      ~ ([($ @),($ @)])
            lc
        ~ ($ @)
            o
     { @ : [], ? - @}*  ).
```

4 The Matching Approach

4.1 Introduction

In order to illustrate the alternative approach, based on matching we return to the `bebop` example given earlier, repeated here:

```
c1:   O[ b ] N[ e ] X[ b ] X[ o ] X[ p ]
c2:   O[ b ] N[ e ] O[ b ] N[ o ] X[ p ]
c3:   X[ b ] X[ e ] O[ b ] N[ o ] X[ p ]
```

Here an instance of 'X[' is a constraint violation, so c2 is the best candidate. By counting, one can see that c2 has one violation, while c1 and c3 each have 3. By matching, one can see that all candidates have a violation in position 13, but c1 and c3 also have violations in positions not corresponding to violations in c2. As long the positions of violations line up in this manner, it is possible to construct a finite state filter to rule out candidates with a non-minimal number of violations. The filter will take the set of candidates, and subtract from that set all strings that are similar, except that they contain additional constraint violations.

Given the approach of marking up constraint violations introduced earlier, it is possible to construct such a matching filter. Consider again the 'bebop' example. If the violations are marked, the candidates of interest are:

```
O[   b ] N[   e ] X[ @ b ] X[ @ o ] X[ @ p ]
O[   b ] N[   e ] O[   b ] N[   o ] X[ @ p ]
X[ @ b ] X[ @ e ] O[   b ] N[   o ] X[ @ p ]
```

For the filter, we want to compare alternative mark-ups for the same input string. Any other differences between the candidates can be ignored. So the first step in constructing the filter is to eliminate everything except the markers and the original input. For the syllable structure example, finding the original input is easy since it never gets changed. For the "bebop" example, the filter first constructs:

```
  b   e @ b @ o   @   p
  b   e   b   o   @   p
@ b @ e   b   o   @   p
```

Since we want to rule out candidates with at least one more constraint violation than necessary, we apply a transducer to this set which inserts at least one more marker. This will yield an infinite set of bad candidates each of which

has at least two markers and with one of the markers coming directly before the final 'p'.

In order to use this set of bad candidates as a filter, brackets have to be reinserted. But since the filter does not care about positions of brackets, these can be inserted randomly. The result is the set of all strings with at least two markers, one of the markers coming directly before the final 'p', and arbitrary brackets anywhere. This set includes the two candidates c1 and c3 above. Therefore, after applying this filter only the optimal candidate survives. The three steps of deleting brackets, adding extra markers and randomly reinserting brackets are encoded in the `add_violation` macro given in fig. 4.

The application of an OT constraint can now be defined as follows, using an alternative definition of the optimality operator:

```
macro(Cands oo Constraint,
          Cands
            o
   mark_violation(Constraint)
            o
     ~ range(Cands
               o
        mark_violation(Constraint)
               o
          add_violation)
            o
      {(@ x []),(? -  @)}* ).
```

Note that this simple approach only works in cases where constraint violations line up neatly. It turns out that for the syllabification example discussed earlier that this is the case. Using the `syllabify` macro given above with this matching implementation of the optimality operator produces a transducer of only 22 states, and can be shown to be exact for all inputs!

4.2 Permutation

In the general case, however, constraint violations need not line up. For example, if the order of constraints is somewhat rearranged as in:

```
parse oo fill_ons oo have_ons
    oo fill_nuc oo no_coda
```

the matching approach is not exact: it will produce wrong results for an input such as 'arts':

```
N[a]D[r]O[t]N[]D[s]       %cf: art@s
N[a]O[r]N[]D[t]O[s]N[]    %cf: ar@ts@
```

Here, the second output should not be produced because it contains one more violation of the `fill_nuc` constraint. In such cases, a limited amount of permutation can be used in the filter to make the marker symbols line up. The `add_violation` filter of fig. 4 can be extended with the following transducer which permutes marker symbols:

```
macro(permute_marker,
 [{[? *,(@ x []),? *,([] x @)],
   [? *,([] x @),? *,(@ x [])]}*,? *]).
```

Greater degrees of permutation can be achieved by composing `permute_marker` several times. For example:[8]

```
macro(add_violation(3),
 {(bracket x []), (? - bracket)}*
               o
      [[? *,([] x @)]+, ? *]
               o
        permute_marker
               o
        permute_marker
               o
        permute_marker
               o
 {([] x bracket), (? - bracket)}*  ).
```

So we can incorporate a notion of 'precision' in the definition of the optimality operator for the matching approach as well, by defining:

```
macro(Cands oo Prec :: Constraint),
        Cands
          o
 mark_violation(Constraint)
          o
   ~ range(Cands
             o
       mark_violation(Constraint)
             o
        add_violation(Prec))
          o
   { (@ x []),(? - @)}*  ).
```

[8] An alternative approach would be to compose the `permute_marker` transducers before inserting extra markers. Our tests, however, show this alternative to be somewhat less efficient.

```
macro(add_violation,
      {(bracket x []), ? - bracket}*     % delete brackets
                     o
           [[? *,([] x @)]+, ? *]        % add at least one @
                     o
      {([] x bracket), ? - bracket}*     % reinsert brackets
     ).
```

Figure 4: Macro to introduce additional constraint violation marks.

The use of permutation is most effective when constraint violations in alternative candidates tend to occur in corresponding positions. In the worst case, none of the violations may line up. Suppose that for some constraint, the input "bebop" is marked up as:

```
c1:   @ b @ e    b    o    p
c2:     b   e  @ b  @ o  @ p
```

In this case, the precision needs to be two in order for the markers in c1 to line up with markers in c2. Similarly, the counting approach also needs a precision of two in order to count the two markers in c1 and prefer this over the greater than two markers in c2. The general pattern is that any constraint that can be treated exactly with counting precision N, can also be handled by matching with precision less than or equal to N. In the other direction, however, there are constraints, such as those in the Prince and Smolensky syllabification problem, that can only be exactly implemented by the matching approach.

For each of the constraint orderings discussed by Prince and Smolensky, it turns out that at most a single step of permutation (i.e. a precision of 1) is required for an exact implementation. We conclude that this OT analysis of syllabification is regular. This improves upon the result of Karttunen (1998). Moreover, the resulting transducers are typically much smaller too. In §5 we present a number of experiments which provide evidence for this observation.

4.3 Discussion

Containment. It might be objected that the Prince and Smolensky syllable structure example is a particularly simple *containment theory* analysis and that other varieties of OT such as *correspondence theory* (McCarthy and Prince, 1995) are beyond the scope of matching.[9] Indeed we have relied on the fact that *Gen* only adds brackets and does not add or delete anything from the set of input symbols. The filter that we construct needs to compare candidates with alternative candidates generated *from the same input.*

If *Gen* is allowed to change the input then a way must be found to remember the original input. Correspondence theory is beyond the scope of this paper, however a simple example of an OT where *Gen* modifies the input is provided by the problem described in §2.2 (from Frank and Satta (1998)). Suppose we modify *Gen* here so that its output includes a representation of the original input. One way to do this would be to adopt the convention that input symbols are marked with a following 0 and output symbols are marked with a following 1. With this convention *Gen* becomes:

```
macro(gen,
 {[(a x [a,0,b,1])*,(b x [b,0,a,1])*],
  [(a x [a,0,a,1])*,(b x [b,0,b,1])*]})
```

Then the constraint against the symbol `a` needs to be recast as a constraint against `[a,1]`.[10] And, whereas above *add_violation* was previously written to ignore brackets, for this case it will need to ignore output symbols

[9] Kager (1999) compares containment theory and correspondence theory for the syllable structure example.

[10] OT makes a fundamental distinction between *markedness* constraints (referring only to the surface) and *faithfulness* constraints (referring to both surface and underlying form). With this mark-up convention, faithfulness constraints might be allowed to refer to both symbols marked with 0 and symbols marked with 1. But note that the `Fill` and `Parse` constraints in syllabification are also considered to be faithfulness constraints since they correspond to epenthesis and deletion respectively.

(marked with a 1). This approach is easily implementable and with sufficient use of permutation, an approximation can be achieved for any predetermined bound on input length.

Locality. In discussing the impact of their result, Frank and Satta (1998) suggest that the OT formal system is too rich in generative capacity. They suggest a *shift in the type of optimization carried out in OT, from global optimization over arbitrarily large representations to local optimization over structural domains of bounded complexity.* The approach of matching constraint violations proposed here is based on the assumption that constraint violations can indeed be compared *locally.*

However, if *locality* is crucial then one might wonder why we extended the local matching approach with global permutation steps. Our motivation for the use of global permutation is the observation that it ensures the matching approach is strictly more powerful than the counting approach. A weaker, and perhaps more interesting, treatment is obtained if locality is enforced in these permutation steps as well. For example, such a weaker variant is obtained if the following definition of `permute_marker` is used:

```
macro(permute_marker,   % local variant
 {? ,[([] x @),?,(@ x [])],
     [(@ x []),?,([] x @)]}* ).
```

This is a weaker notion of permutation than the definition given earlier. Interestingly, using this definition resulted in equivalent transducers for all of the syllabification examples given in this paper. In the general case, however, matching with local permutation is less powerful.

Consider the following artificial example. In this example, inputs of interest are strings over the alphabet $\{b, c\}$. *Gen* introduces an `a` *before* a sequence of `b`'s, or two `a`'s *after* a sequence of `b`'s. *Gen* is given as an automaton in fig. 5. There is only a single constraint, which forbids `a`. It can easily be verified that a matching approach with global permutation using a precision of 1 exactly implements this OT. In contrast, both the counting approach as well as a matching approach based on local permutation can only approximate this OT.[11]

[11]Matching with local permutation is not strictly more powerful than counting. For an example, change *Gen* in this example to: `{[([] x a),{b,c}*],[{b,c}*,([] x [a,a])]}`. This can be exactly implemented by counting with a precision of one. Matching with local permutation, however, cannot exactly implement this case, since markers would need to be permuted across unbounded sequences.

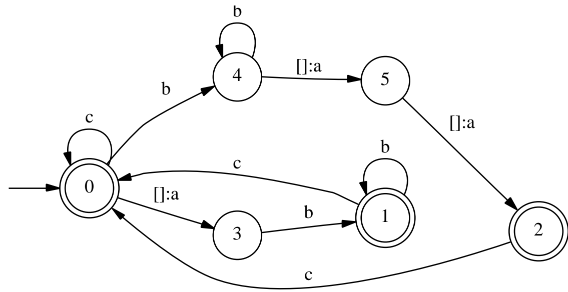


Figure 5: *Gen* for an example for which local permutation is not sufficient.

5 Comparison

In this section we compare the two alternative approaches with respect to accuracy and the number of states of the resulting transducers. We distinguish between *exact* and *approximating* implementations. An implementation is exact if it produces the right result for all possible inputs.

Assume we have a transducer T which correctly implements an OT analysis, except that it perhaps fails to distinguish between different numbers of constraint violations for one or more relevant constraints. We can decide whether this T is exact as follows. T is exact if and only if T is exact with respect to each of the relevant constraints, i.e., for each constraint, T distinguishes between different numbers of constraint violations. In order to check whether T is exact in this sense for constraint C we create the transducer `is_exact(T,C)`:

```
macro(is_exact(T,C),
          T
          o
     mark_violation(C)
          o
   {(? - @) x [], @}*).
```

If there are inputs for which this transducer produces multiple outputs, then we know that `T` is not exact for `C`; otherwise `T` *is* exact for `C`. This reduces to the question of whether `is_exact(T,C)` is ambiguous. The question

of whether a given transducer is ambiguous is shown to be decidable in (Blattner and Head, 1977); and an efficient algorithm is proposed in (Roche and Schabes, 1997).[12] Therefore, in order to check a given transducer T for exactness, it must be the case that for each of the constraints C, is_exact(T,C) is nonambiguous.

If a transducer T is not exact, we characterize the quality of the approximation by considering the maximum length of input strings for which T is exact. For example, even though T fails the exactness check, it might be the case that

```
[? ^,? ^,? ^,? ^,? ^]
        o
        T
```

in fact is exact, indicating that T produces the correct result for all inputs of length ≤ 5.

Suppose we are given the sequence of constraints:

```
have_ons >> fill_ons >> parse
   >> fill_nuc >> no_coda
```

and suppose furthermore that we require that the implementation, using the counting approach, must be exact for all strings of length ≤ 10. How can we determine the level of precision for each of the constraints? A simple algorithm (which does not necessarily produce the smallest transducer) proceeds as follows. Firstly, we determine the precision of the first, most important, constraint by checking exactness for the transducer

```
gen oo P :: have_ons
```

for increasing values for P. As soon as we find the minimal P for which the exactness check succeeds (in this case for P=0), we continue by determining the precision required for the next constraint by finding the minimal value of P in:

```
gen oo 0 :: have_ons oo P :: fill_ons
```

We continue in this way until we have determined precision values for each of the constraints. In this case we obtain a transducer with 8269 states implementing:

```
gen oo 0 :: have_ons
    oo 1 :: fill_ons
    oo 8 :: parse
    oo 5 :: fill_nuc
    oo 4 :: no_coda
```

In contrast, using matching an exact implementation is obtained using a precision of 1 for the fill_nuc constraint; all other constraints have a precision of 0. This transducer contains only 28 states.

The assumption in OT is that each of the constraints is universal, whereas the constraint *order* differs from language to language. Prince and Smolensky identify nine interestingly different constraint orderings. These nine "languages" are presented in table 2.

In table 3 we compare the size of the resulting automata for the matching approach, as well as for the counting approach, for three different variants which are created in order to guarantee exactness for strings of length ≤ 5, ≤ 10 and ≤ 15 respectively.

Finally, the construction of the transducer using the matching approach is typically much faster as well. In table 4 some comparisons are summarized.

6 Conclusion

We have presented a new approach for implementing OT which is based on matching rather than the counting approach of Karttunen (1998). The matching approach shares the advantages of the counting approach in that it uses the finite state calculus and avoids off-line sorting and counting of constraint violations. We have shown that the matching approach is superior in that analyses that can only be approximated by counting can be exactly implemented by matching. Moreover, the size of the resulting transducers is significantly smaller.

We have shown that the matching approach along with global permutation provides a powerful technique technique for minimizing constraint violations. Although we have only applied this approach to permutations of the Prince & Smolensky syllabification analysis, we speculate that the approach (even with local permutation) will also yield exact implementations for most other OT phonological analyses. Further investigation is needed here, particularly with recent versions of OT such as cor-

[12] We have adapted the algorithm proposed in (Roche and Schabes, 1997) since it fails to treat certain types of transducer correctly; we intend to provide details somewhere else.

id	constraint order
1	`have_ons` ≫ `fill_ons` ≫ `no_coda` ≫ `fill_nuc` ≫ `parse`
2	`have_ons` ≫ `no_coda` ≫ `fill_nuc` ≫ `parse` ≫ `fill_ons`
3	`no_coda` ≫ `fill_nuc` ≫ `parse` ≫ `fill_ons` ≫ `have_ons`
4	`have_ons` ≫ `fill_ons` ≫ `no_coda` ≫ `parse` ≫ `fill_nuc`
5	`have_ons` ≫ `no_coda` ≫ `parse` ≫ `fill_nuc` ≫ `fill_ons`
6	`no_coda` ≫ `parse` ≫ `fill_nuc` ≫ `fill_ons` ≫ `have_ons`
7	`have_ons` ≫ `fill_ons` ≫ `parse` ≫ `fill_nuc` ≫ `no_coda`
8	`have_ons` ≫ `parse` ≫ `fill_ons` ≫ `fill_nuc` ≫ `no_coda`
9	`parse` ≫ `fill_ons` ≫ `have_ons` ≫ `fill_nuc` ≫ `no_coda`

Table 2: Nine different constraint orderings for syllabification, as given in Prince and Smolensky, chapter 6.

Method	Exactness	Constraint order								
		1	2	3	4	5	6	7	8	9
matching	exact	29	22	20	17	10	8	28	23	20
counting	≤ 5	95	220	422	167	10	240	1169	2900	4567
counting	≤ 10	280	470	1667	342	10	420	8269	13247	16777
counting	≤ 15	465	720	3812	517	10	600	22634	43820	50502

Table 3: Comparison of the matching approach and the counting approach for various levels of exactness. The numbers indicate the number of states of the resulting transducer.

respondence theory. Another line of further research will be the proper integration of finite state OT with non-OT phonological rules as discussed, for example, in papers collected in Hermans and van Oostendorp (1999) .

Finally, we intend also to investigate the application of our approach to syntax. Karttunen (1998) suggests that the Constraint Grammar approach of Karlsson et al. (1995) could be implemented using lenient composition. If this is the case, it could most probably be implemented more precisely using the matching approach. Recently, Oflazer (1999) has presented an implementation of Dependency syntax which also uses lenient composition with the counting approach. The alternative of using a matching approach here should be investigated.

References

Meera Blattner and Tom Head. 1977. Single-valued a-transducers. *Journal of Computer and System Sciences*, 15(3):328–353.

Mark T. Ellison. 1994. Phonological derivation in optimality theory. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, pages 1007–1013, Kyoto.

Robert Frank and Giorgio Satta. 1998. Optimality theory and the computational complexity of constraint violability. *Computational Linguistics*, 24:307–315.

Dale Gerdemann and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway.

Ben Hermans and Marc van Oostendorp, editors. 1999. *The Derivational Residue in Phonological Optimality Theory*, volume 28 of *Linguistik Aktuell/Linguistics Today.* John Benjamins, Amsterdam/Philadelphia.

C. Douglas Johnson. 1972. *Formal Aspects of Phonological Descriptions.* Mouton, The Hague.

René Kager. 1999. *Optimality Theory.* Cambridge UP, Cambridge, UK.

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

Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila. 1995. *Constraint Gram-*

Method	Exactness	Constraint order								
		1	2	3	4	5	6	7	8	9
matching	exact	1.0	0.9	0.9	0.9	0.8	0.7	1.5	1.3	1.1
counting	≤ 5	0.9	1.7	4.8	1.6	0.5	1.9	10.6	18.0	30.8
counting	≤ 10	2.8	4.7	28.6	4.0	0.5	4.2	83.2	112.7	160.7
counting	≤ 15	6.8	10.1	99.9	8.6	0.5	8.2	336.1	569.1	757.2

Table 4: Comparison of the matching approach and the counting approach for various levels of exactness. The numbers indicate the CPU-time in seconds required to construct the transducer.

mar: A Language-Independent Framework for Parsing Unrestricted Text. Mouton de Gruyter, Berlin/New York.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–238.

Lauri Karttunen. 1995. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, M.I.T. Cambridge Mass.

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

Lauri Karttunen. 1998. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara.

André Kempe and Lauri Karttunen. 1996. Parallel replacement in the finite-state calculus. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, Copenhagen, Denmark.

Charles Kisseberth. 1970. On the functional unity of phonological rules. *Linguistic Inquiry*, 1:291–306.

John McCarthy and Alan Prince. 1995. Faithfulness and reduplicative identity. In Jill Beckman, Laura Walsh Dickey, and Suzanne Urbanczyk, editors, *Papers in Optimality Theory*, pages 249–384. Graduate Linguistic Student Association, Amherst, Mass. University of Massachusetts Occasional Papers in Linguistics 18.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Kemal Oflazer. 1999. Dependency parsing with an extended finite state approach. In *37th Annual Meeting of the Association for Computational Linguistics*, pages 254–260.

Alan Prince and Paul Smolensky. 1993. Optimality theory: Constraint interaction in generative grammar. Technical Report TR-2, Rutgers University Cognitive Science Center, New Brunswick, NJ. MIT Press, To Appear.

Emmanuel Roche and Yves Schabes. 1997. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, Mass.

Bruce Tesar. 1995. *Computational Optimality Theory*. Ph.D. thesis, University of Colorado, Boulder.

Gertjan van Noord and Dale Gerdemann. 1999. 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.

Gertjan van Noord. 1997. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*, pages 87–108. Springer Verlag. Lecture Notes in Computer Science 1260.

Gertjan van Noord. 1999. FSA6 reference manual. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/˜vannoord/Fsa/.

Markus Walther. 1996. OT simple – a construction-kit approach to optimality theory implementation. ROA-152-1096.

Multi-Syllable Phonotactic Modelling

Anja Belz
CCSRC, SRI International
23 Millers Yard, Mill Lane
Cambridge CB2 1RQ, UK
anjab@cam.sri.com

Abstract

This paper describes a novel approach to constructing phonotactic models. The underlying theoretical approach to phonological description is the multi-syllable approach in which multiple syllable classes are defined that reflect phonotactically idiosyncratic syllable subcategories. A new finite-state formalism, OFS Modelling, is used as a tool for encoding, automatically constructing and generalising phonotactic descriptions. Language-independent prototype models are constructed which are instantiated on the basis of data sets of phonological strings, and generalised with a clustering algorithm. The resulting approach enables the automatic construction of phonotactic models that encode arbitrarily close approximations of a language's set of attested phonological forms. The approach is applied to the construction of multi-syllable word-level phonotactic models for German, English and Dutch.

1 Introduction

Finite-state models of phonotactics have been used in automatic language identification (Zissman, 1995; Belz, 2000), in speech recognition (Carson-Berndsen, 1992; Jusek et al., 1994; Jusek et al., 1996; Carson-Berndsen, 2000), and optical character recognition, among other applications. While statistical models (n-gram or Markov models) are derived automatically from data, their symbolic equivalents are usually constructed in a painstaking manual process, and — because based on standard single-syllable phonological analyses — tend to overgeneralise greatly over a language's set of wellformed phonological strings. This paper describes methods that enable the automatic construction of symbolic phonotactic models that are more accurate representations of phonological grammars.

The underlying theoretical approach to phonological description is the *Multi-Syllable Approach* (Belz, 1998; Belz, 2000). Syllable phonotactics vary considerably not only in correlation with a syllable's position within a word, but also with other factors such as position relative to word stress. Analyses based on multiple syllable classes defined to reflect such factors can more accurately account for the phonologies of natural languages than analyses based on a single syllable class.

Object-Based Finite State Modelling (previously described in Belz, 2000) is used as an encoding, construction and generalisation tool, and facilitates *Language-Independent Prototyping*, where incompletely specified generic models are constructed for groups of languages and subsequently instantiated and generalised automatically to fully specified, language-specific models using data sets of phoneme strings from individual languages. The theory-driven (manual) component in this construction method is restricted to specifying the maximum possible ways in which syllable phonotactics may differ in a family of languages, without hard-wiring the differences into the final models. The actual construction of models for individual languages is a data-driven process and is done automatically.

Sets of German, English and Dutch syllables were used extensively in the research described in this paper, both as a source of evidence in support of the multi-syllable approach (Section 2) and as data in automatic phonotactic model construction (Section 4). All syllable sets were derived from sets of fully syllabified, phonetically transcribed forms collected from the lexical database CELEX (Baayen et al., 1995). CELEX contains compounds and phrases as well as single words. Phonological words were defined as any phonetic sequence with a single primary stress marker, and all other entries were disregarded.

2 Multi-Syllable Phonotactics

The multi-syllable approach works on the assumption that single-syllable approaches cannot adequately capture the phonological grammars of natural languages, because they fail to account for the significant syllable-based phonotactic variation resulting from a range of factors that is evident in natural languages, and consequently overgeneralise greatly.

Single-syllable analyses. The traditional view is that all syllables in a language share the same structure and compositional constraints which can

	German		English		Dutch	
	all	unique (%)	all	unique (%)	all	unique (%)
Initial	3,806	624 (16.4%)	6,177	2,657 (43.01%)	5,476	947 (17.29%)
Medial	3,832	358 (9.34%)	3,149	344 (10.92%)	5,446	723 (13.28%)
Final	7,040	2,133 (30.3%)	6,750	2,132 (31.59%)	7,279	1,786 (24.54%)
Monosyllables	5,114	855 (16.72%)	7,265	2,963 (40.78%)	5,641	718 (12.73%)
TOTAL	10,606	3,970 (37.43%)	14,333	8,096 (56.49%)	11,448	4,174 (36.46%)

Table 1: Syllable set sizes and number of syllables unique to each set (position).

be captured by a single analysis. In many languages, however, the sets of word-initial and/or word-final consonant clusters differ significantly from other consonantal clusters (Goldsmith, 1990, p. 107ff, lists several examples from different languages). Such idiosyncratic clusters have been treated as 'terminations', 'appendices', or as 'extrasyllabic' (Goldsmith, 1990), and integrated along with syllables at the word-level. Similar, apparently irregular phenomena occur in correlation with tone and stress, and the first and last vocalic segments in phonological words are often analysed as 'extratonal' and 'extrametrical'. However, such apparent irregularities are not restricted to the beginnings and ends of phonological words, and the phonotactics of syllables are affected by a range of factors other than position, which are difficult if not impossible to account for by the notion of extrasyllabicity.

Three problematic issues arise in single-syllable analyses. Firstly, if a phonotactic model assumes a single syllable class for a language, and if the language has idiosyncratic word-initial and word-final phonotactics, then the set of possible phonological words that the model encodes is necessarily too large, and includes words that form systematic (rather than accidental) gaps in the languages. Secondly, if extrasyllabicity is used to account for phonotactic idiosyncracies, then the resulting theory of syllable structure fails to account for everything that it is intended to account for, and is forced to integrate constituents that are not syllables (the *extra*syllabic material) at the word level. Thirdly, the notion of extrasyllabicity only works for cases where phonemic material can be segmented off adjacent syllables (most easily done at the beginnings and ends of words), and cannot be used to account for syllable-internal variation. The alternative offered by multi-syllable analyses is to make the universal assumption that position, stress and tone (among other factors) will result in variation in syllable phonotactics that are not necessarily restricted to any particular part of words, and to account for such variation systematically by the use of different syllable classes.

Related approaches. The idea to discriminate between different syllable types, classified by word position and position with respect to the stressed syllable has been explored and utilised in previous research, for example in FSA-based phonotactic models, typed formalisms, and in stochastic production rule grammars. Carson-Berndsen (1992) uses two separate FSAs to encode the phonotactics of full and reduced syllables, and Jusek et al. (1994) distinguish between stressed and unstressed syllables. In a typed feature system of morpho-phonology, Mastroianni and Carpenter (1994) define subtypes of the general type *syllable*.

The most closely related existing research is that presented by Coleman and Pierrehumbert (1997). The paper examines different possibilities for using a probabilistic grammar for English words to model native speakers' acceptability judgments. The production rule grammar encodes the phonotactics of English monosyllabic and bisyllabic words. Different probability distributions over paths in derivation trees are investigated which model likelihood of acceptability to native speakers, rather than likelihood of occurrence. To build a grammar that accounts for interactions among onsets and rhymes, location with respect to the word edge and word stress patterns, six syllable types are distinguished which reflect possible combinations of the features strong, weak, initial and final. The subsyllabic constituents onset and rhyme are similarly marked for stress and position.

The present research extends existing work on syllable subclasses by applying the multi-syllable approach systematically to model the entire phonotactics of languages, and by using it for language-independent prototyping (see Section 3.3 below).

Position-correlated phonotactic variation. Table 1 shows statistics for sets of monosyllabic words and initial, medial and final syllables in CELEX. For each language and each syllable set, the table shows the size of the set (e.g. there are 3,806 different initial German syllables in CELEX), and the size of its subset of syllables that do not occur in any other set (e.g. 624 out of 3,806 initial German syllables, or 16.4%, only occur word-initially). For all three languages, the figures show significant differences between the sets of syllables that can occur in the four different positions and their unique

		Medial	Final	Mono
German:	Initial	2,619 (0.52)	1,466 (0.16)	1,392 (0.18)
	Medial		1,928 (0.22)	1,185 (0.15)
	Final			3,873 (0.47)
English:	Initial	1,860 (0.25)	1,920 (0.17)	2,266 (0.20)
	Medial		1,787 (0.22)	1,008 (0.11)
	Final			3,576 (0.34)
Dutch:	Initial	3,594 (0.49)	2,764 (0.28)	3,003 (0.37)
	Medial		3,279 (0.35)	2,428 (0.28)
	Final			4,320 (0.50)

Table 2: Intersections and set similarities for German, English and Dutch syllables (position).

	German		English		Dutch	
	all	unique (%)	all	unique (%)	all	unique (%)
Stressed	8,919	2,977 (33.37%)	9,399	5,280 (56.18%)	9,934	3,484 (35.07%)
Pretonic	989	30 (3.03%)	3,201	1,362 (42.55%)	1,780	71 (3.99%)
Posttonic	5,897	388 (6.58%)	4,754	670 (14.09%)	5,960	517 (8.67%)
Plain	6,819	229 (3.36%)	6,020	944 (15.68%)	6,662	176 (2.64%)
TOTAL	10,598	3,624 (34.20%)	14,333	8,256 (57.60%)	11,443	4,248 (37.12%)

Table 3: Syllable set sizes and number of syllables unique to each set (stress).

subsets. In German and Dutch, final syllables are particularly idiosyncratic, with 30.3% and 24.54%, respectively, not occurring in any other position. In English, all syllable sets except the medial syllables display a high degree of idiosyncracy. Table 2 shows the size of the intersections between the syllable sets, and the more objective measure of set similarity in brackets[1]. In German and Dutch, the similarity between initial and medial syllables, and between final and monosyllables is particularly high. The similarity between the least similar of syllable sets is much greater in Dutch than in either English or German. In English, only the final and monosyllables display any significant similarity. Average set similarity is highest in Dutch (0.37), followed by German (0.28), and English (0.21).

Stress-correlated phonotactic variation. Table 3 shows analogous statistics for phonotactic variation correlated with word stress. Set sizes and unique subset sizes are shown for the set of syllables that carry primary stress (stressed), those immediately preceding stress (pretonic), those immediately following stress (posttonic), and all others (plain). In all three languages, the set of stressed syllables has least in common with other sets. In English, this is closely followed by the pretonic syllables. The average percentage of syllables unique to a set is highest in English, followed by Dutch and then German.

[1] Set similarity here is the standard measure of the size of the intersection over the size of the union of two sets S_1 and S_2, or $|S_1 \cap S_2|/|S_1 \cup S_2|$ (not defined for $S_1 = S_2 = \emptyset$).

These statistics show not only that there is significant syllable-level variation in the phonotactics of all three languages, but also that the simple strategy of subdividing the set of all syllables on the basis of position and stress succeeds in capturing at least some of this variation. If a high percentage of syllables in one subcategory do not occur in any other, then distinguishing this syllable subcategory in a phonotactic model will help reduce overgeneralisation.

3 Encoding, Construction and Generalisation of Phonotactic Models

3.1 Object-Based Finite-State Modelling

The OFS Modelling formalism was used as a tool for encoding, constructing and generalising phonotactic models in the research described in Section 4. OFS Modelling consists of three main components, (i) a representation formalism, (ii) a mechanism for automatic model construction, and (iii) mechanisms for model generalisation. Brief summaries of the components that were used in the research described in this paper are given here (for full details see Belz, 2000).

Underlying OFS Modelling is a set of assumptions about linguistic description that shares many of the fundamental tenets of declarative phonology (Bird, 1991, for example). This set of assumptions includes a strictly non-derivational, non-transformational and constraint-based approach to linguistic description, and the principle of constraint inviolability.

The OFS formalism is a declarative, monostratal finite-state representation formalism that is intuitively readable, facilitates the automatic data-driven construction of models, and permits the integration of available prior, theoretical knowledge. The derivations (trees or brackettings) defined by OFS models correspond to context-free derivations with a limited tree depth or degree of nesting of brackets. This means that in OFS models (unlike in other normal forms for regular grammars), rules (hence expansions or brackets) can, if appropriately defined, systematically correspond to standard linguistic objects, the reason why the formalism is called *object-based*.

OFS Model $O = (N, T, P, n+1)$			
n:	O_0^n	$\Rightarrow$	ω_0^n
n-1:	O_0^{n-1}	$\Rightarrow$	ω_0^{n-1}
	O_1^{n-1}	$\Rightarrow$	ω_1^{n-1}
	...		
	O_m^{n-1}	$\Rightarrow$	ω_m^{n-1}
...			
1:	O_0^1	$\Rightarrow$	ω_0^1
	O_1^1	$\Rightarrow$	ω_1^1
	...		
	O_l^1	$\Rightarrow$	ω_l^1
0:	O_0^0	$\Rightarrow$	ω_0^0
	O_1^0	$\Rightarrow$	ω_1^0
	...		
	O_p^0	$\Rightarrow$	ω_p^0

Figure 1: Notational convention for OFS models.

OFS Models. The OFS representation formalism is essentially a normal form for regular sets. OFS models can be interpreted in the same way as standard production rule grammars, but are subject to a set of additional constraints. An OFS model O is denoted $(N, T, P, n+1)$, where N is a finite set of non-terminal objects O_j^i, $0 \leq i \leq n$, and T is a finite set of terminals. P is an ordered finite set of n sets of productions $O_j^i \Rightarrow \omega_j^i$, where $O_j^i \in N$, and for $i > 0$, ω_j^i is a regular expression[2] over symbols $O_h^g \in N, i > g$, whereas for $i = 0$, ω_j^i is a set of strings[3] from T^*. An OFS model O has n levels, or sets of production rules, and each rule $O_j^i \Rightarrow \omega_j^i$ is uniquely associated with one of the levels. The nth set of production rules is a singleton set $\{O_0^n \Rightarrow \omega_0^n\}$, and O_0^n is interpreted as the start symbol. The notational convention adopted for OFS models is as shown in Figure 1.

Definition 1 *OFS Model*
An OFS model O is a 4-tuple $(N, T, P, n+1)$, where N is a finite set of nonterminals O_j^i, $0 \leq i \leq n$, $O_0^n \in N$ is the start symbol, T is a finite set of terminals, $n+1$ denotes the number of levels in the model, and $P =$

$$\{ \{O_0^n \Rightarrow \omega_0^n\}, \{O_0^{n-1} \Rightarrow \omega_0^{n-1}, O_1^{n-1} \Rightarrow \omega_1^{n-1}, \dots O_m^{n-1} \Rightarrow \omega_m^{n-1}\}, \dots \{O_0^1 \Rightarrow \omega_0^1, O_1^1 \Rightarrow \omega_1^1, \dots O_l^1 \Rightarrow \omega_l^1\}, \{O_0^0 \Rightarrow \omega_0^0, O_1^0 \Rightarrow \omega_1^0, \dots O_p^0 \Rightarrow \omega_p^0\} \},$$

where each rule $O_j^i \Rightarrow \omega_j^i$ is uniquely associated with one of the levels, ω_j^0 is a set of strings from T^*, $\omega_j^i, i > 0$, is a regular expression over objects $O_h^g \in N, i > g$.

Each rule $O \Rightarrow \omega$ in an OFS model corresponds to a set of strings which will be referred to as an object set or class, where O is the name of the object. The production rules in OFS models will also be referred to as object rules.

OFS models thus differ from standard production rule grammars in three ways. Firstly, RHSs of rules above level 0 are arbitrary regular expressions[4]. Secondly, terminals from T are restricted to appearing in the RHSs of rules at level 0 (mostly to facilitate automatic model construction, see below). Thirdly, OFS models are limited in their representational power to the finite-state domain by the constraints that the RHSs of rules in rule sets at level $i > 0$ are regular expressions over non-terminals that appear only in the LHSs of rules in rule sets at levels $g < i$. That this limits representational power to the regular languages can be seen from the fact that all non-terminals O_j^i in the RHS of the single top-level rule can be substituted iteratively with the RHSs of the corresponding rules $O_j^i \Rightarrow \omega_j^i$. This iteration terminates after a finite time because there is a finite number of levels in the model, and at this point the RHS of the top-level rule contains only non-terminals, i.e. *is* a regular expression, hence represents a regular language.

Unlike other normal forms for regular production-rule grammars (such as left-linear and right-linear

[2] In the regular expressions in this paper, r^* denotes any number of repetitions of r, r^+ denotes at least one repetition of r, and $r + e$ denotes the disjunction of r and e.

[3] The string sets in level 0 RHSs are actually implemented more efficiently as finite automata.

[4] Other formalisms for linguistic analysis have permitted full regular expressions in the RHSs of rules. For instance, in syntactic grammars, the recursive nature of some types of coordination has been modelled with right-recursive regular expressions (e.g. in GPSG).

sets of production rules), OFS models enable the definition of production rules and hence derivations that can, if appropriately defined, correspond to standard linguistic objects and constituents (not possible in linear grammars). Through the association of rules with a finite number of levels, OFS models permit the definition of grammars that encode sets of context-free derivations up to a maximum depth equal to the number of levels in the model.

The fact that non-terminal strings are in OFS models restricted to the lowest level, facilitates the combined theory and data driven construction of models. Uninstantiated models can be defined, that encode what is known in advance about the structural regularities of the object to be modelled in levels above 0, and have under-specified level 0 RHSs that are subsequently instantiated on the basis of data sets of examples of the object to be modelled. OFS Modelling also has a generalisation procedure which can be used to generalise fully instantiated OFS models. Each of these mechanisms is described in turn over the following paragraphs.

Uninstantiated OFS Models. In fully specified OFS models (as defined in the preceding section), the right-hand sides (RHSs) of production rules at level i are regular expressions for $i > 0$, and string sets for $i = 0$. This separation makes it simple to construct incompletely specified models, or *prototype OFS models*, where the RHSs of level 0 rules are pattern descriptions rather than strings sets. Level 0 RHSs in prototype models have the form $O_i^0 \Rightarrow S_i$, where O_i^0 is the name of the object, and S_i is a set former $\{x : \mathbf{v}x\mathbf{w} \in D, P_1, P_2, \ldots P_n\}$, where $\mathbf{v}, \mathbf{w}$ are concatenations of variables, D refers to any given finite data set of strings, and $P_i, 1 \leq i \leq n$ are properties of the variables in $\mathbf{v}$ and $\mathbf{w}$.

Instantiation of Prototype OFS Models. The OFS instantiation procedure takes a prototype OFS model M for some linguistic object and a data set D of example members of the corresponding object class and proceeds as follows. For each level 0 rule $O_i^0 \Rightarrow S_i$ in M, and for each element x of D, all substrings of x that match S_i are collected. The resulting set of substrings becomes the new RHS of rule O_i^0. After instantiation, level 0 rules whose RHS is the empty set are removed, as are rules at higher levels whose RHSs contain non-terminals that can no longer be expanded by any of the production rules in M.

Object-Set Generalisation. Instantiated OFS models can be generalised by object-set (OS) generalisation, where pairs of level 0 object sets are compared on the basis of a standard set similarity measure sim for two finite sets D_1 and D_2 (not defined for $D_1 = D_2 = \emptyset$): $sim(D_1, D_2) = |D_1 \cap D_2|/|D_1 \cup D_2|$. The OS-generalisation procedure takes a fully specified OFS model M and a given similarity threshold τ, and, applying a simple clustering algorithm, merges all object sets that have a similarity value sim matching or exceeding τ. That is, the OS-generalisation procedure measures the similarity between all pairs of level 0 sets, and all pairs that match or exceed the threshold end up in the same cluster. Finally, the old object names (non-terminals) in the RHSs of object rules at levels above 0 are replaced with the LHSs of the corresponding new merged object rule, while all object rules that now have identical RHSs are in turn merged. In this way, generalisation 'percolates' upwards through the levels of the model.

Determining an appropriate value for the similarity threshold τ is not unproblematic. It could be set in relation to the average similarity value in an instantiated model (individually for each prototype instantiation), but this approach would obscure the similarities that object-set generalisation (in particular in conjunction with LIP) is intended to exploit. The whole point of object-set generalisation for language-independent prototypes is that it will merge a different number of level 0 object classes in different prototype instantiations, creating different final, language-specific OFS models. If τ is set in proportion to the average similarity between level 0 classes, then this difference is reduced, and the resulting models will tend to retain the same number of level 0 object classes from the prototype. For example, if the above prototype model $Word$ is instantiated to a data set from a language that has phonotactics which differ only between stressed and unstressed syllables, then all similarity values between stressed syllable classes regardless of their position within a word, and between all posttonic, pretonic and plain syllables classes (again, regardless of position), will be very high. The average similarity value will therefore also be high. If τ is set in relation to this high average, not all unstressed and all stressed syllable classes, respectively, will be merged, because not all syllable classes can exceed average similarity.

Average similarity is a language-specific property, and so is the number of syllable classes similar enough to be merged for a given τ value. For different generalised instantiations of the same prototype model to be comparable, object-set generalisation must have been carried out for each of them with the same τ value.

The threshold τ is best regarded as a variable parameter to the OS-generalisation procedure that can be used to control the degree to which a generalised OFS model will fit the data: the higher τ, the more closely the model will fit the data, and the less it will generalise over it. This is particularly appropriate in phonotactic modelling, because phonotactics seeks to encode not just the set of attested words, but also

Prototype OFS Model $Syllable = (\{Syllable, Onset, Peak, Coda\}, T, P, 2)$			
1:	*Syllable*	$\Rightarrow$	*Onset Peak Coda*
0:	*Onset*	$\Rightarrow$	$\{x \mid xay \in D, x \in CONSONANTS^*, a \in VOWELS\}$
	Peak	$\Rightarrow$	$\{x \mid yxz \in D, x \in VOWELS^+, y, z \in CONSONANTS^*\}$
	Coda	$\Rightarrow$	$\{x \mid yax \in D, x \in CONSONANTS^*, a \in VOWELS\}$

Figure 2: Simple prototype OFS model for syllable-level phonotactics.

æz, æʃ, ɑːsk, æsp, æs, æt, ɛt, ɔːk, ɔːks, ɑːnts, ɔː, ɔːz, æks, aɪ, aɪz, beɪ, bɑː, bɑːz, beɪb, bæk, bæks, siː, kæb, ʧɛə∗, ʧɛəd, sɪnʧ, sɪnʧt, kliːv, dɛf, diːl, djuːst, dʌvz, drɑːfts, dwɛld, faɪ, frɛt, gəʊld, grɒt, kwɪd, splæt, sprɪŋ, stræps, stʌn

Figure 3: Small data set of English monosyllabic words.

OFS Model $Syllable = (\{Syllable, Onset, Peak, Coda\}, T, P, 2)$			
1:	*Syllable*	$\Rightarrow$	*Onset Peak Coda*
0:	*Onset*	$\Rightarrow$	{ ϵ, b, s, k, st, f, d, ʧ, kl, dj, dr, dw, fr, g, gr, kw, spl, spr, str }
	Peak	$\Rightarrow$	{æ, ɑː, ɛ, ɔː, aɪ, eɪ, iː, ɛə, ʌ, əʊ, ɒ, ɪ, uː }
	Coda	$\Rightarrow$	{ ϵ, b, s, k, st, f, d, z, ʃ, sk, sp, ks, nts, ∗, nʧ, nʧt, v, l, vz, fts, ld, t, ŋ, ps, n }

Figure 4: Syllable-level phonotactic OFS model instantiated with set of English monosyllables.

OFS Model $Syllable = (\{Syllable, Onset_Coda, Peak, \}, T, P, 2)$			
1:	*Syllable*	$\Rightarrow$	*Onset_Coda Peak Onset_Coda*
0:	*Onset_Coda*	$\Rightarrow$	{ ϵ, b, s, k, st, f, d, ʧ, kl, dj, dr, dw, fr, g, gr, kw, spl, spr, str, z, ʃ, sk, sp, ks, nts, ∗, nʧ, nʧt, v, l, vz, fts, ld, t, ŋ, ps, n }
	Peak	$\Rightarrow$	{æ, ɑː, ɛ, ɔː, aɪ, eɪ, iː, ɛə, ʌ, əʊ, ɒ, ɪ, uː }

Figure 5: OFS model of Figure 4 generalised with $\tau \leq 0.19$.

unattested, but wellformed words (often called 'accidental' gaps), while excluding only illformed words (or 'systematic' gaps). There is no objective dividing line between idiosyncratic and systematic gaps, and setting τ can be used as one way of controlling the degree of conservativeness in generalising over the set of attested words.

3.2 Example

As an illustration, consider the following example construction of a simple OFS model for syllable-level phonotactics (the constraints that hold on the possible phoneme sequences within syllables)[5]. The prototype OFS model constructed in the first step (Figure 2) encodes the standard assumption that the syllable-level phonotactics in different languages can be appropriately modelled by interpreting syllables as a sequence of consonantal phonemes (onset), followed by a sequence of vocalic phonemes (peak), and another sequence of consonantal phonemes (coda).

In the second construction step, a data set of English monosyllabic words (Figure 3) is used to instantiate the prototype OFS model. The instantiation procedure constructs an OFS model with new level 0 RHSs as shown in Figure 4. During OS-generalisation, *sim* values are computed for each pair of level 0 object sets. The only pairwise intersection that is non-empty (hence the only non-zero *sim* value) in this example is that between the sets *Coda* and *Onset* (*sim* = 0.19), which are merged if OS-generalisation is applied to OFS model *Syllable* with $\tau \leq 0.19$, resulting in the simpler, more general OFS model shown in Figure 5.

3.3 Language-Independent Prototyping

Language-independent prototyping (LIP) as a general approach to linguistic description seeks to define generic models that restrict — in some linguistically meaningful way — the set of grammars or descriptions that can be inferred from data. OFS modelling can be used as an implementational tool for LIP. Language-independent prototype OFS models can be defined by specifying a maximal number of objects and corresponding production rules such that when the prototype is instantiated and generalised with data sets from individual languages, dif-

[5] The example model is not intended to be a realistic phonotactic model, but is provided here merely as an illustration of the techniques outlined above.

Prototype OFS Model $Word = (N, M, P, 2)$			
1:	$Word$	$\Rightarrow$	S_mon_st +
			S_mon_pl +
			$(S_ini_st\ S_fin_po)$ +
			$(S_ini_st\ S_med_po\ S_med_pl^*\ S_fin_pl)$ +
			$(S_ini_pr\ S_fin_st)$ +
			$(S_ini_pr\ S_med_st\ S_fin_po)$ +
			$(S_ini_pr\ S_med_st\ S_med_po\ S_med_pl^*\ S_fin_pl)$ +
			$(S_ini_pl\ S_med_pl^*\ S_med_pr\ S_fin_st)$ +
			$(S_ini_pl\ S_med_pl^*\ S_med_pr\ S_med_st\ S_fin_po)$ +
			$(S_ini_pl\ S_med_pl^*\ S_med_pr\ S_med_st\ S_med_po\ S_med_pl^*\ S_fin_pl)$
0:	S_mon_st	$\Rightarrow$	$\{x : {}'x \in D, x \in (M \backslash \{-\})^*\}$
	S_mon_pl	$\Rightarrow$	$\{x : x \in D, x \in (M \backslash \{-,'\})^*\}$
	S_ini_st	$\Rightarrow$	$\{x : {}'x - w \in D, x \in (M \backslash \{-\})^*\}$
	S_ini_pr	$\Rightarrow$	$\{x : x -' vw \in D, x, v \in (M \backslash \{-\})^*\}$
	S_ini_pl	$\Rightarrow$	$\{x : x - u -' vw \in D, x, v \in (M \backslash \{-\})^*\}$
	S_med_st	$\Rightarrow$	$\{x : v -' x - w \in D, x \in (M \backslash \{-\})^*\}$
	S_med_pr	$\Rightarrow$	$\{x : u - x -' vw \in D, x, v \in (M \backslash \{-\})^*\}$
	S_med_po	$\Rightarrow$	$\{x : u'v - x - w \in D, x, v \in (M \backslash \{-\})^*\}$
	S_med_pl	$\Rightarrow$	$\{x : (u'y - v - x - w \in D) \vee (u - x - v -' w \in D), x \in (M \backslash \{-\})^*\}$
	S_fin_st	$\Rightarrow$	$\{x : w -' x \in D, x \in (M \backslash \{-\})^*\}$
	S_fin_po	$\Rightarrow$	$\{x : w'v - x \in D, x, v \in (M \backslash \{-\})^*\}$
	S_fin_pl	$\Rightarrow$	$\{x : w'v - u - x \in D, x, v \in (M \backslash \{-\})^*\}$

Figure 6: Prototype OFS model for multi-syllable word-level phonotactics.

ferent object sets will be deleted and merged for different languages, resulting in different final, instantiated and generalised OFS models. In the following section, a language-independent phonotactic prototype OFS model is instantiated to surprisingly different OFS models for three closely related languages.

4 Multi-Syllable Phonotactic Models for German, English and Dutch

When applied to modelling multi-syllable word-level phonotactics, LIP with OFS Modelling means defining the maximum possible number of syllable classes that may be subject to different phonotactic constraints in a given group of languages. The exact set of syllable classes depends on the group of languages the prototype is intended to cover as well as the desired amount of generalisation over data (in general, a model that distinguishes only two syllable classes will generalise more than a model that distinguishes three or more classes, given the same data). The prototype presented in this section is intended to cover German, English and Dutch, and takes into account only phonological factors (syntactic factors such as word category which can also affect phonotactics are not taken into account). Two phonological factors are modelled: position of a syllable within a word, and position of a syllable relative to primary word stress.

For this modelling task, the LIP approach is implemented by constructing an OFS prototype model in which syllable classes reflecting all possible different combinations of position within a word and relative to stress are defined as level 0 uninstantiated object rules, and all possible ways in which the corresponding objects can be combined to form words are defined as higher-level object rules. No prior assumptions about where phonotactic variation occurs is hardwired into the model. Instead, the maximal ways in which phonotactics may vary in a group of languages is encoded. The idea is that prototype instantiation and OS-generalisation with data sets of phonological words from different languages will result in different final, instantiated phonotactic models.

4.1 Language-Independent Prototype OFS Model for Multi-syllable Phonotactics

The prototype model shown in Figure 6 distinguishes between twelve syllable classes which correspond to all possible combinations of position within a word and position relative to primary stress ($'$ marks primary stress, $-$ is the syllable separator, and S = syllable). As before, the set of all syllables is divided into four classes on the basis of position (*mon* = monosyllabic, *ini* = initial, *med* = medial, *fin* = final), each of which is divided further into four subclasses on the basis of stress (*st* = stressed, *pr* = pretonic, *po* = posttonic, *pl* = plain). This results in a total of 12 possible syllable cat-

	German		English		Dutch	
	all	unique (%)	all	unique (%)	all	unique (%)
Set_mon_st	5,028	849 (16.89%)	7,254	2,958 (40.77%)	5,641	719 (12.75%)
Set_mon_pl	1,813	1 (0.06%)	11	5 (45.45%)	0	- (-)
Set_ini_st	3,658	527 (14.41%)	3,345	409 (12.23%)	5,258	772 (14.68%)
Set_ini_pr	707	18 (2.55%)	2,560	1,328 (51.88%)	1,346	49 (3.64%)
Set_ini_pl	1,628	19 (1.17%)	1,495	437 (29.23%)	1,252	28 (2.24%)
Set_med_st	2,527	92 (3.64%)	1,600	90 (5.63%)	3,907	282 (7.22%)
Set_med_pr	618	12 (1.94%)	916	30 (3.28%)	1,026	26 (2.53%)
Set_med_po	2,518	66 (2.62%)	1,415	65 (4.59%)	3,296	185 (5.61%)
Set_med_pl	2,220	28 (1.26%)	1,156	82 (7.09%)	2,897	36 (1.24%)
Set_fin_st	4,261	822 (19.29%)	3,376	583 (17.27%)	4,972	803 (16.15%)
Set_fin_po	4,354	413 (9.49%)	4,141	882 (21.3%)	4,525	460 (1.02%)
Set_fin_pl	3,716	166 (4.47%)	2,635	306 (11.61%)	3,820	101 (2.64%)
TOTAL	10,598	3,013 (28.42%)	14,333	7,175 (50.06%)	11,443	3,461 (30.25%)

Table 4: Sizes of Level 0 object sets resulting from instantiations, and syllables unique to each set.

egories[6]. D is the data set given in instantiation, and M the corresponding set of terminals (here, the phonemic symbols that occur in D). The RHS of the level 1 object rule encodes all possible ways in which the twelve syllable classes can theoretically combine to form words. The prototype model is language-independent, because not all syllable classes will exist in all languages (e.g. a language where primary stress is always on the first syllable would not have classes of word-initial pretonic or plain syllables), and OS-generalisation will create different new syllable classes, depending on which classes are most similar in a given language.

4.2 Prototype Model Instantiations

Table 4 shows the sizes of the different level 0 object sets resulting from OFS model instantiations to the German, English and Dutch word sets derived from CELEX (the syllable sets are far too large to be shown in their entirety). In all three languages, the largest syllable set is the set of stressed monosyllables, and the smallest is the set of medial pretonic syllables[7]. Table 4 also shows (in the same format as in Section 2) the number of syllables in each syllable class that do not occur in any of the other classes.

In German and Dutch, percentages of unique syllables are significantly lower than in the classes reflecting position only and stress only that were shown in Section 2, indicating that some of the classes may not be worth distinguishing in phonotactic models. In English, however, the higher percentages of unique syllables are not far behind those shown previously, indicating that most of the twelve syllable classes in the prototype are worth distinguishing.

Some correlation is evident between the size of a set and the percentage of unique syllables it contains. In German, average syllable set size is $2,754$ and the average percentage of unique syllables is 6.48%. Five syllable sets are of above average size, and four of these also have above-average percentages of unique syllables. Seven syllable sets are below average in size, and non of these have above-average percentages of unique syllables. In English, the picture is not as straightforward. Average syllable set size is $2,717$, and average percentage of unique syllables is 18.62%. Of the four sets of above-average size, two have above-average, and two have below-average, percentages of unique syllables. Of the seven English syllable sets of below-average size (the set of plain monosyllables is disregarded again for English and Dutch), two have above-average, and five have below-average percentages of unique syllables. Finally, in Dutch, average set size is $3,449$ and average percentage of unique syllables is 6.33%. Four of the six above-average sized sets also have above-average percentages of unique syllables, while all of the below-average sized sets also have below-average percentages of unique syllables. However, there is no complete correlation, with some of the largest sets having very small percentages of unique syllables, and vice versa.

4.3 OS-Generalisation of Models

As is clear from the instantiation results presented in the preceding section, some syllable classes contain such low percentages of unique syllables that it is not worth distinguishing them as a separate class. OS-generalisation of models can be used to merge the most similar classes and reduce the number of classes that the model distinguishes.

[6] Not $4 \times 4 = 16$ classes, because some classes cannot exist (e.g. there is no such thing as a posttonic initial syllable).

[7] Disregarding the set of plain monosyllables of which there were no examples in the Dutch section of CELEX, and only a very small number in the English section.

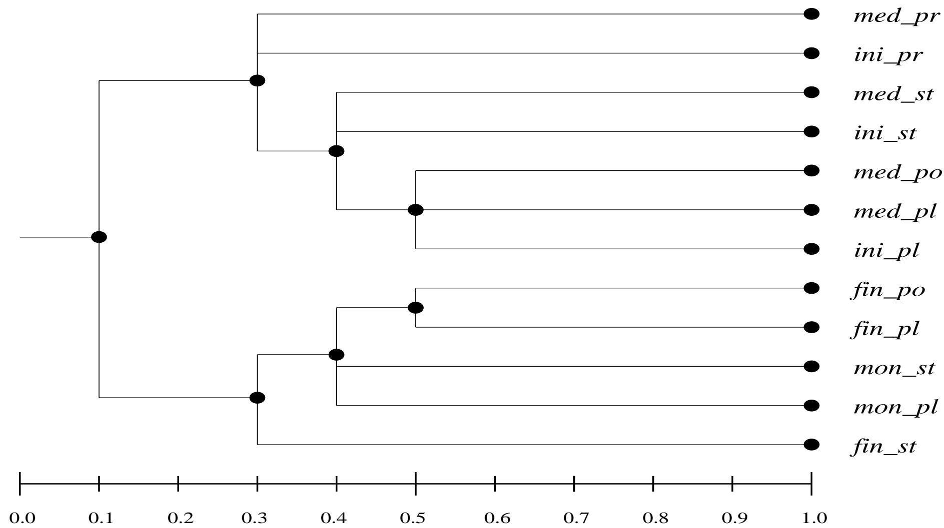


Figure 7: Cluster tree for German syllable sets.

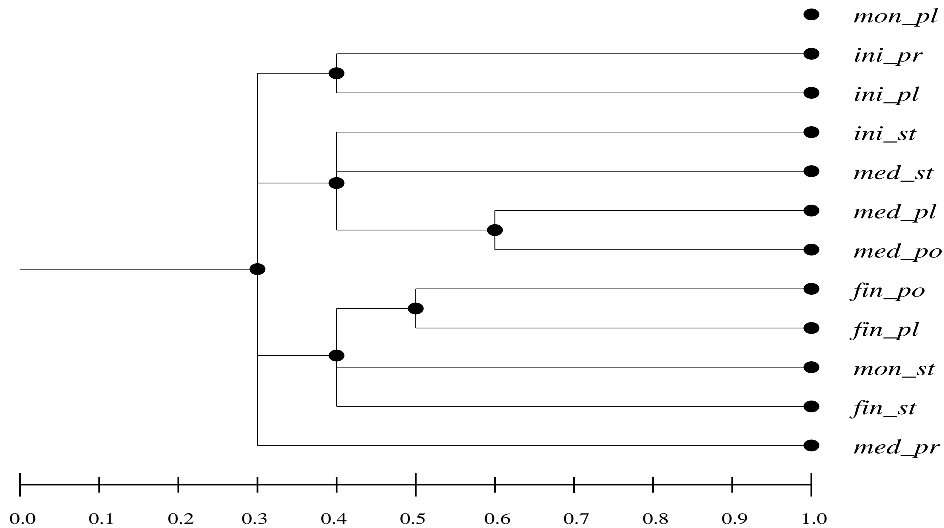


Figure 8: Cluster tree for Dutch syllable sets.

4.3.1 Generalisation of Multi-Syllable OFS Model for German

Figure 7 shows the cluster tree for the German syllable sets produced by carrying out OS-generalisation for $\tau = 0.1..1.0$ in increments of 0.1. Each node in the tree shows at which τ values the original syllable sets at the leaves dominated by the node were merged. The tree reveals a very neat picture for German. 0.56 is the highest τ value between any syllable class pair, so for $\tau \geq 0.6$ no classes are merged. $\tau = 0.5$ results in two clusters, one containing final unstressed syllables, the other initial and medial unstressed syllables. At $\tau = 0.4$, all monosyllables are added to the final syllable class, and one more medial and one more initial class to the set of initial and medial syllables. At $\tau = 0.3$, all monosyllables and final syllables on the one hand, and all initial and medial syllables on the other, are merged. Setting τ lower makes no difference until it is set below 0.2, at which point all of the original syllable classes are merged into a single set.

This shows clearly that in German the distinction between monosyllables and final syllables on the one hand, and between initial and medial syllables on the other, is very strongly marked (preserved even when τ is set as low as 0.2). This distinction is thus marked far more strongly than the unstressed/stressed division (which is more commonly encoded in DFA models of German phonotactics), which disappears at $\tau = 0.4$ (in fact, even earlier, at $\tau = 0.47$).

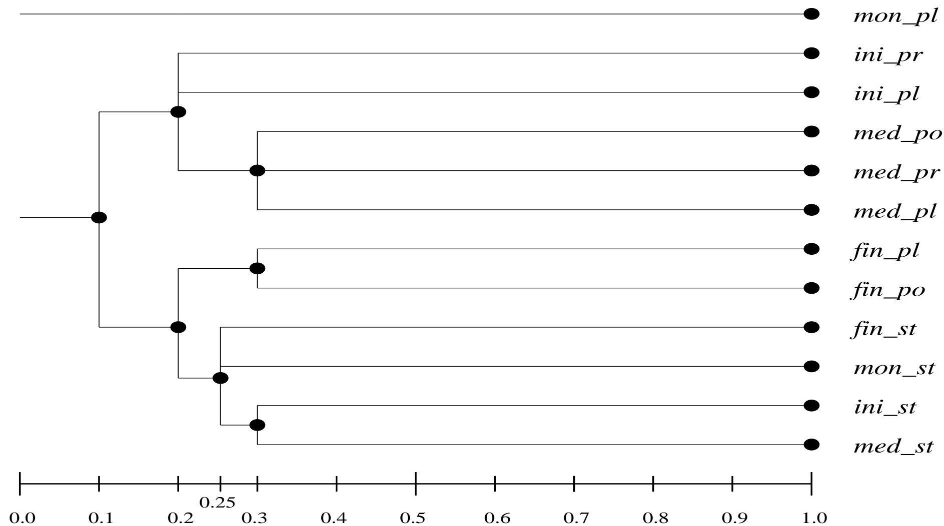


Figure 9: Cluster tree for English syllable sets.

4.3.2 Generalisation of Multi-Syllable OFS Model for Dutch

The cluster tree for Dutch (Figure 8) also reveals an important division between final and monosyllables on the one hand, and initial and medial syllables on the other. However, it is not as clearly marked as in German. There is a point ($\tau = 0.4$) when all final and monosyllables are in the same cluster, but this is not the case for the initial and medial syllables, which form subclusters that are correlated with stress. The medial plain and posttonic syllable sets are merged with each other at $\tau = 0.6$, and with the initial stressed and medial stressed syllables at $\tau = 0.4$. But there is no greater similarity between this cluster and the cluster of inital pretonic and plain syllables (formed at $\tau = 0.4$) than there is between it and the cluster of final and monosyllables. All three are merged into a single cluster at $\tau = 0.3$.

4.3.3 Generalisation of Multi-Syllable OFS Model for English

In the cluster tree for English (Figure 9), there are clusters clearly correlated with stress and clusters clearly correlated with position. At $\tau = 0.3$ three clusters are formed, one containing all medial syllable sets except the stressed medial syllables, another containing all final syllable sets except the stressed final syllables, and the third containing two stressed syllable sets. At $\tau = 0.25$, all stressed syllables together form one cluster. However, at $\tau = 0.2$, two unstressed syllable sets are added to this cluster, while all the remaining unstressed sets form the other large cluster. Thus, in English, both stress and position are strong determinants of phonotactic variation, but differences resulting from stress are more pronounced than those resulting from position.

4.4 Discussion

The LIP approach implemented with OFS Modelling proceeds in three steps. First, the factors likely to produce phonotactic idiosyncracy (stress and position in the above examples), and the constituents to be used in the analysis (syllables only in the above examples), are decided, and a prototype model is constructed on this basis. This prototype distinguishes as many objects at level 0 as there are possible combinations of factors and lowest-level constituents. All ways in which these objects can combine to form higher-level constituents are encoded at the corresponding higher levels in the model.

In the second step, the prototype is instantiated with data sets from different languages. The degree to which the instantiated models generalise over the given data is determined by the number of constituents and subcategories of constituents distinguished in the prototype. As an example, consider the different degrees to which three models that discriminate different numbers of syllable classes generalise over given data. All three models define words as sequences of syllables, and syllables as sequences of phonemes. The first model has only one syllable class, the second distinguishes four classes reflecting position in a word, and the third is the same as the model presented in the preceding section, i.e. distinguishes twelve syllable classes. After instantiation with the same data set of German phonological word forms from CELEX used previously, the three models will encode supersets of the data set that generalise over it to different degrees. Looking at subsets of words of the same length gives some impression of the differences. For instance, model 1 encodes 10,598 monosyllabic German words (the total number of

different syllables in the data), whereas models 2 and 3 encode only 6,841 monosyllables (the actual number of monosyllabic words in CELEX). The following table shows the number of bisyllabic words each model encodes.

Model	Bisyllabic words
(1) *Syll Syll*	1.12×10^8
(2) *Syll_ini Syll_fin*	2.67×10^7
(3) (*Syll_ini_pr Syll_fin_st*)+ (*Syll_ini_st Syll_fin_po*)	1.89×10^7
Attested forms	7.09×10^4

Model 3 permits about 266 times as many bisyllabic word forms as there are in CELEX, model 2 encodes 1.4 times as many as model 3, and model 1 encodes 4.2 times as many as model 2. Thus, through progressively finer grained subcategories of syllables, progressively closer approximations of the set of attested forms can be achieved.

However, doing this in an indiscriminate, language-independent way may produce some syllable classes that are very similar. With OS-generalisation, the most similar classes can be merged, so that only strongly marked differences are preserved. However, setting τ to any specific value is problematic. Producing cluster trees with a range of τ values can give some idea of important class distinctions, and can be used as a basis for determining an appropriate τ value. τ can further be motivated by different linguistic assumptions and the intended purpose of the generalised models. Generalising different instantiations of the same prototype for the same τ value, makes it possible to compare the relative markedness of phonotactic variation in different languages.

5 Summary and Further Research

This paper described how OFS modelling and the multi-syllable approach can be combined with language-independent prototyping to create a method for designing phonotactic models that (i) facilitates automatic model construction, (ii) produces models that are arbitrarily close approximations of the set of wellformed phonological words in a given language, and (iii) provides a generalisation method with control over the degree to which final models fit given data. Extensions of the approach currently under investigation include stochastic OFS models, and the integration of OFS models into finite-state syntactic grammars.

References

R. H. Baayen, R. Piepenbrock, and L. Gulikers, editors. 1995. *The CELEX Lexical Database (CD-ROM).* Linguistic Data Consortium, University of Pennsylvania, Philadelphia.

Anja Belz. 1998. An approach to the automatic acquisition of phonotactic constraints. In T. Mark Ellison, editor, *Proceedings of SIGPHON '98: The Computation of Phonological Contraints*, pages 35–44.

Anja Belz. 2000. *Computational Learning of Finite State Models for Natural Language Processing.* Ph.D. thesis, School of Cognitive and Computing Sciences, University of Sussex.

S. Bird, editor. 1991. *Declarative Perspectives on Phonology*, volume 7 of *Edinburgh Working Papers in Cognitive Science.* Centre for Cognitive Science, University of Edinburgh.

Julie Carson-Berndsen. 1992. An event-based phonotactics for German. Technical Report ASL-TR-29-92/UBI, Fakultät für Linguistik und Literaturwissenschaft, University of Bielefeld.

Julie Carson-Berndsen. 2000. Finite state models, event logics and statistics in speech recognition. In *Philosophical Transactions of the Royal Society*, volume A 358. Royal Society. In press.

J. S. Coleman and J. Pierrehumbert. 1997. Stochastic phonological grammars and acceptability. In *Proceedings of the Third Meeting of the ACL Special Interest Group in Computational Phonology, SIGPHON '97*, pages 49–56.

John A. Goldsmith. 1990. *Autosegmental and Metrical Phonology.* Blackwell, Cambridge.

Jusek, Fink, Kummert, Sagerer, Berndsen, and Gibbon. 1994. Detektion unbekannter Wörter mit Hilfe phonotaktischer Modelle. In W. Kropatsch and H. Bischof, editors, *Mustererkennung '94, 16. DAGM-Symposium und 18. Workshop der ÖAGM Wien*, pages 238–245.

A. Jusek, A. Fink, F. Kummert, and G. Sagerer. 1996. Automatically generated models for unknown words. In *Proceedings of the Sixth Australian International Conference on Speech Science and Technology*, pages 301–306.

Michael Mastroianni and Bob Carpenter. 1994. Constraint-based morpho-phonology. In *Proceedings of the First Meeting of the ACL Special Interest Group in Computational Phonology, SIGPHON '94.*

Marc A. Zissman. 1995. Language identification using phoneme recognition and phonotactic language modelling. In *Proceedings of ICASSP '95*, volume 5, pages 3503–3506.

Taking Primitive Optimality Theory Beyond the Finite State

Daniel M. Albro
Linguistics Department
UCLA

Abstract

Primitive Optimality Theory (OTP) (Eisner, 1997a; Albro, 1998), a computational model of Optimality Theory (Prince and Smolensky, 1993), employs a finite state machine to represent the set of active candidates at each stage of an Optimality Theoretic derivation, as well as weighted finite state machines to represent the constraints themselves. For some purposes, however, it would be convenient if the set of candidates were limited by some set of criteria capable of being described only in a higher-level grammar formalism, such as a Context Free Grammar, a Context Sensitive Grammar, or a Multiple Context Free Grammar (Seki et al., 1991). Examples include reduplication and phrasal stress models. Here we introduce a mechanism for OTP-like Optimality Theory in which the constraints remain weighted finite state machines, but sets of candidates are represented by higher-level grammars. In particular, we use multiple context-free grammars to model reduplication in the manner of Correspondence Theory (McCarthy and Prince, 1995), and develop an extended version of the Earley Algorithm (Earley, 1970) to apply the constraints to a reduplicating candidate set.

1 Introduction

The goals of this paper are as follows:

- To show how finite-state models of Optimality Theoretic phonology (such as OTP) can be extended to deal with non-finite state phenomena (such as reduplication) in a principled way.
- To provide an OTP treatment of reduplication using the standard Correspondence Theory account.
- To extend the Earley chart parsing algorithm to multiple context free grammars (MCFGs).

The basic idea of this approach is to begin with a non-finite-state description of the space of acceptable candidates (*e.g.,* candidates with some sort of reduplication inherent in them, or candidates which are the outputs of a syntactic grammar), and to repeatedly intersect the high-level grammar representing those candidates with finite state machines representing constraints. The intersection operation is one of weighted intersection (where only the set of lowest-weighted candidates survive) in order to model Optimality Theory, and will make use of a modified version of the Earley parsing algorithm.

There are at least two alternative approaches to that which we will propose here: to abandon finite state models altogether and move to uniformly higher-level approaches (*e.g.,* Tesar (1996)), or to modify finite state models minimally to allow for (perhaps limited) reduplication (*e.g.,* Walther (2000)). The first of these alternative approaches deals with context free grammars alone, so it would not be able to model reduplicative effects. Besides this, it seems preferable to stick with finite-state approaches as far as possible, because phonological effects beyond the finite state seem quite rare. The second of these approaches seems reasonable in itself, but it is not suited for the type of analyses for which the approach laid out here is designed. In particular, Walther's approach is tied to One-Level Phonology, a theory which limits itself to surface-true generalizations, whereas the approach here is designed to model Optimality Theory—a system with violable constraints—and in particular Correspon-

dence Theory. Tesar's approach as well, while it is a model of Optimality Theory, does not seem suited to Correspondence Theory. A final argument for using this approach, in preference to one similar to Walther's approach, is that it can be extended to cover other non-finite-state areas of phonology, such as phrasal stress patterns, with no modification to the basic model.

2 Quick Overview of OTP

2.1 Optimality Theory

Optimality Theory (OT), of which OTP is a formalized computational model, is structured as follows, with three components:

1. **Gen:** a procedure that produces infinite surface candidates from an underlying representation (UR)
2. **Con:** a set of constraints, defined as functions from representations to integers
3. **Eval:** an evaluation procedure that, in succession, winnows out the candidates produced by **Gen.**

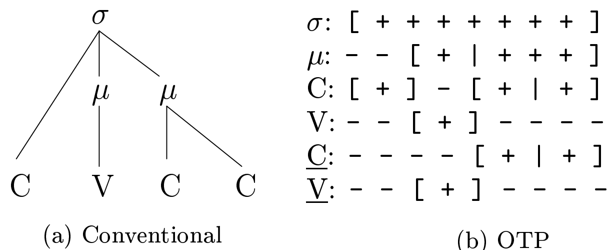
So OT is a theory that deals with potentially infinite sets of phonological representations. The OT framework does not by itself specify the character of these representations, however.

2.2 Primitive Optimality Theory (OTP)

The components of OT, as modeled by OTP (see Eisner (1997a), Eisner (1997b), Albro (1998)):

1. **Gen:** a procedure that produces from an underlying representation a finite state machine that represents all possible surface candidates that contain that UR (always an infinite set)
2. **Con:** a set of constraints definable in a restricted formalism—internally represented as Weighted Deterministic Finite Automata (WDFAs) which accept any string in the representational alphabet. The weights correspond to constraint violations. The weights passed through when accepting a string are the violations incurred by that string.
3. **Eval:** the following procedure, where I represents the input FSM produced by Gen, and M is a machine representing the output set of candidates:

   $M \leftarrow I$
   for all $C_i \in$ **Con**, taken in rank order **do**
     $M \leftarrow$ intersection of M with C_i
     Remove non-optimal paths from M
     Zero out weights in M
   end for



(a) Conventional (b) OTP

Figure 1: OTP Representation

Representations in OTP are gestural scores using symbols from the set $\{-, +, [,], |\}$. See Figure 1 for an example. This figure shows a CVCC syllable in a conventional notation, and also in OTP notation. The OTP notation is slightly more complex, though, in that it also shows an underlying form for the syllable. The overlap relation of the conventional notation's association lines is expressed in the OTP notation by the presence of constituent interiors ("+") in the same vertical slice through the diagram. This same-time-slice-membership relation is also used to show correspondence. Thus we see from this diagram that the surface "CVCC" syllable corresponds to underlying "VCC," and that the initial "C" does not correspond to any underlying segment. Note that tiers with no special marking are used to represent the surface level of representation, and underlined tiers are used to represent the underlying level of representation.

3 Handling Reduplication: Overview

3.1 Overview

Finite State Machines are useful in phonology because it is possible to take any two finite

state machines, each of which represents a set of strings, and perform an *intersection* operation on them. The resulting machine represents the intersection of the two sets of strings. For example, this allows us to use constraints represented as FSMs to limit a candidate set.

Although we would sometimes like to characterize the candidate sets using CFGs or MCFGs, it must be kept in mind that these formalisms do not have the property of being intersectable with each other. Thus, in OTP terms, it would not be possible to represent the constraints as CFGs or MCFGs. However, there is a way out: it is possible to intersect an FSM with a CFG or an MCFG.

Based on the above, an approach to handing reduplication in phonology becomes clear—we start with an MCFG that enforces reduplicative identity, then intersect it with the input FSM (produced by **Gen**), then the constraint FSMs, as before. The hard part, then, is to come up with an efficient FSM-intersection algorithm for MCFGs which also deals correctly with weighted FSMs.

3.2 MCFGs

A grammar formalism that is midway between CFGs and CSGs in expressive power, an MCFG is like a CFG except that categories may rewrite to tuples of strings instead of rewriting to just one string as usual. It should be noted that MCFGs have been shown (van Vugt, 1996) to be equivalent to string-valued attribute grammars with only s-attributes, relational grammars, and top-down tree-to-string transducers, so we could use any one of these grammars to provide a candidate space. As an example of an MCFG, here's a simple MCFG for the language $\{ww | w \in \{0,1\}^+\}$ (the language of total reduplication):

$$\begin{array}{lcl} S & \rightarrow & A_0\, A_1 \\ A & \rightarrow & (1,1) \\ & | & (0,0) \\ & | & (0\, A_0, 0\, A_1) \\ & | & (1\, A_0, 1\, A_1) \end{array}$$

The nonterminals of this grammar are S, which has arity 1, and A, which has arity 2. The right-hand sides of the productions include notations such as A_0, which indicate the placement of each part of the tuple-yield of any category. Here, A_0 and A_1 are the two parts of the single category A, so a rule like $A \rightarrow (0\, A_0, 0\, A_1)$ indicates that A rewrites to $0\, 0\, A$, with the actual strings arranged in a tuple with a 0 preceding the first part of A in the first half of the pair, and 0 preceding the second part of A in the second half of the pair.

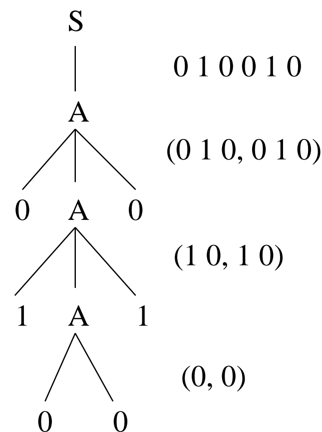


Figure 2: Derivation of "010010"

This grammar is in the normal form required by the algorithms presented here. This normal form can be characterized as follows:

> For any category C of arity greater than 1, the category may appear in the right hand side of a production only if the right hand side refers to each element of C exactly once.

A derivation of the string "010010" in this grammar would go as follows: S rewrites as $A_0 A_1$, that is, to the concatenation of the string-yield of the two parts of A. From here, A_0 and A_1 must both come from the parts of a single one of the four productions for A. A then rewrites to $(0\, A_0, 0\, A_1)$, making, for example, the value of A_0 in the S production be $(0\, A_0)$. A then rewrites to $(1\, A_0, 1\, A_1)$, so S reduces to $01\, A_0\, 01\, A_1$. Finally, A rewrites to $(0, 0)$, leaving the value of S as 010010. This derivation is illustrated in Figure 2, the left side of which depicts the derivation tree, while its right side shows (from the bottom up) the string-yield of each non-terminal (shown just below and to the right of it).

3.3 Representation of Reduplicative Forms in OTP

OTP constraints are inherently local—they can only refer to overlap or non-overlap of interiors or edges in an instant of time. Therefore, to enforce correspondences between forms, they must be juxtaposed so as to occur in the same time-slices. In OTP, correspondence between the surface and underlying forms is established by using one set of tiers for the surface form (each tier represents either a feature or a type of prosodic constituent) and another corresponding set for the underlying form. For example, the tier *son* might specify the distribution of the surface feature "sonorant", while the tier $\underline{son}$ would specify its underlying correspondent. Elements of those tiers placed in the same time-slice are considered to be in correspondence with one another. In order to create correspondence between two portions of the same surface form, then, we need to somehow have them simultaneously juxtaposed so as to appear in the same segments of time and separated in time as they will be on the surface. This is accomplished by a representational trick: in the example of reduplication, a copy of the reduplicant's surface form is placed in a special set of tiers within the base:

SL:	BASE	RED_2
UL:	UR_1	UR_2
RL:	RED_1	—

— or —

SL:	RED_2	BASE
UL:	UR_2	UR_1
RL:	—	RED_1

In these representations **SL** stands for the surface level of representation, **UL** for the underlying level, and **RL** for the special reduplicant level (the place where a copy of the reduplicant is kept). UR_1 and UR_2 are identical in the input, and RED_1 and RED_2 need to be kept identical by other means. The means chosen here is to use an MCFG enforcing the identity. BASE-RED correspondence constraints operate upon RED_1 while templatic and general surface well-formedness constraints operate upon RED_2. An example of this sort of representation might help here. Suppose that there are two surface tiers, *C* and *V*. Then a form such as [CV+CVC] (with CV prefixing reduplication, assuming that the base is CVC, and with the underlying form RED+/VC/) might be represented as follows:

C:	[	+	]	-	-	-	[	+	]	-	[	+	]
V:	-	-	[	+	]	-	-	-	[	+	]	-	-
$\underline{C}$:	-	-	-	-	[	+	]	-	-	-	[	+	]
$\underline{V}$:	-	-	[	+	]	-	-	-	[	+	]	-	-
$\underline{\underline{C}}$:	-	-	-	-	-	-	[	+	]	-	-	-	-
$\underline{\underline{V}}$:	-	-	-	-	-	-	-	-	[	+	]	-	-
INS:	[	+	]	-	-	-	[	+	]	-	-	-	-
DEL:	-	-	-	-	[	+	]	-	-	-	-	-	-
RDEL:	-	-	-	-	-	-	-	-	-	-	[	+	]
RED:	[	+	+	+	+	+	]	-	-	-	-	-	-
BASE:	-	-	-	-	-	-	[	+	+	+	+	+	]

Note here that the special *BASE* and *RED* tiers indicate the portions of the surface forms that are the base and reduplicant, and that the reduplicant level of representation (that is, the level that holds the copy of the reduplicant used for correspondence) is present on the tiers labeled with double underlines. The *INS* tier represents a time-discrepancy between the levels of representation where time does not exist on the underlying level (so the period of time taken up by the initial C in the surface reduplicant and base doesn't correspond to anything in the underlying level), and the *DEL* tier represents time that does not exist on the surface level, so the time taken up by the final C in the underlying form of the reduplicant does not correspond to anything on the surface. The *RDEL* tier is a mirror of the contents of the *DEL* tier in the surface reduplicant, and thus represents time that does not exist in the special reference copy of the reduplicant. This representation allows us to notice that the reduplicant fits a CV template — the left edge of it is aligned with a surface C, the right edge with a surface V, and there are no other segments within it. (The relevant OTP constraints to reinforce this would be "RED[$\rightarrow$ C[," "]RED $\rightarrow$]V," "]C $\perp$ C[$\perp$ RED," and "]C $\perp$ V[$\perp$ RED," if highly ranked and in that order.)

In terms of translating these representations to finite state machines (or to strings), we use the alphabet $\{-, +, [,], |\}$, so that each FSM edge is labeled with a member of this alphabet. This representation differs from that of earlier accounts of OTP, in that the FSM edges in those accounts represented entire time slices, whereas

an edge in this representation represents a single tier in a time slice. As an example, the representation of:

C: [$+^*$]
V: - - -

is as shown in Figure 3, where the "C" and "V" labels are not part of the representation, but just there to ease reading.

3.4 The Grammar Used

The grammar used here is a bit complicated, but the important thing to note about it is that it generates exactly the set of possible OTP output forms in which the special reduplicant reference level of representation contains an exact copy of the surface reduplicant, placed within the time-duration of the base. The grammar for a situation in which there are two surface tiers appears in Figure 4. Extending this grammar to other numbers of tiers is straightforward. The constituents of this grammar are as follows:

S The start symbol.

Non Non-reduplicating material (such as non-reduplicating morphemes) before and/or after the reduplicating material.

SSR The surface tiers in a time-slice.

UR The underlying tiers in a time-slice.

MRD The reduplicant reference-level tiers in a time-slice where the tiers must contain the value $-$ (that is, outside of the base, which is the only place where the reduplicant level is used).

Rd/Rd1/Rd2 The reduplicating part of an utterance.

BDR A right-facing boundary (allows anything to be in the surface tiers during its time-slice, and copies the right-facing half of that material into the reduplicant).

BDL A left-facing boundary (see *BDR*).

B The surface tiers in a time-slice plus identical material in the reduplicant tiers. Thus *B* represents an item in the reduplicant plus its copy in the special reduplicant reference level.

The remaining non-terminals define different values for the INS, DEL, RDEL, RED, and BASE tiers, where INS and DEL are as defined in Albro (1998), RDEL represents time that does not exist in the reduplicant, RED represents the reduplicant (as a morpheme boundary), and BASE represents the base as a morpheme boundary:

NBR represents the state of not being in the base or the reduplicant.

RLE represents the left edge of the reduplicant.

RRE represents the right edge of the reduplicant.

BLE represents the left edge of the base.

BRE represents the right edge of the base.

RB represents a boundary between a reduplicant and a base, where the reduplicant comes first.

BR represents the reverse of *RB*.

RED represents the inside of the reduplicant.

BASE represents the inside of the base.

In this grammar any given time-slice will be defined as *SSR* or the first component of one of the *B* categories, followed by *UR*, followed by *MRD* or the second component of one of the *B* categories, followed by one of the *NBR*, etc., categories.

4 The Earley Algorithm

The Earley algorithm is an efficient chart parsing method. Chart parsing can be seen as a method for taking the intersection of a string or FSM with a CFG (later, an MCFG). Here we take a CFG as a 4-tuple $\langle V, N, P, S \rangle$ where V represents the set of terminals in the grammar, N represents the set of non-terminals, P represents the set of productions, and $S \in N$ is the start symbol. In the definitions to follow, α, β, and γ represent arbitrary members of $(V \cup N)^*$, A and C represent arbitrary members of N, a and b represent arbitrary members of V, p represents an arbitrary member of P, and the indices i, j, and k represent positions within the input string to be parsed, numbered as in Figure 5.

In the standard definition, a member of the chart is a 3-tuple $(i, C \rightarrow \alpha \bullet \beta, j)$, where i represents the position at the beginning of the input

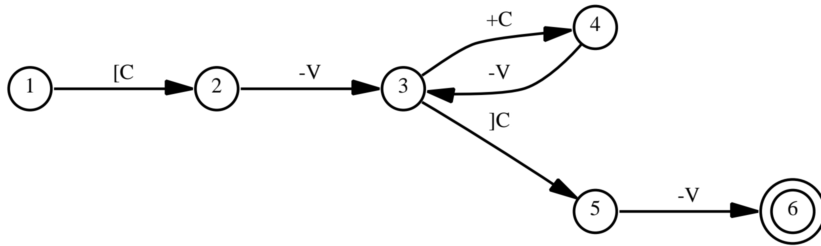


Figure 3: FSM Representation Used Here

string covered by α and j represents the position at the end of the covered portion of the string. The parsing operation in the standard definition, which parses a single input string, is defined as a closure via the following three inference rules of a chart initially consisting of $(0, S \rightarrow \bullet\alpha, 0)$:

predict: $\frac{(i,C\rightarrow\alpha\bullet A\beta,j)}{(j,A\rightarrow\bullet\gamma,j)}$ if $A \rightarrow \gamma \in P$ (if γ begins with a terminal, that terminal must be the symbol at position j in the input string)

scan: $\frac{(i,C\rightarrow\alpha\bullet a\beta,j)}{(i,C\rightarrow\alpha a\bullet\beta,j+1)}$ if a is the symbol after j

complete: $\frac{(i,C\rightarrow\alpha\bullet A\beta,j)\ (j,A\rightarrow\gamma\bullet,k)}{(i,C\rightarrow\alpha A\bullet\beta,k)}$

The input string is recognized if the chart contains an element $(0, S \rightarrow \alpha\bullet, n)$, where n is the final position of the input string.

5 Extending Earley

The algorithm presented so far just checks to see whether a particular string exists in a grammar. In order for it to be useful for our purposes, the following extensions must be made:

1. Intersection with an FSM, not just a string
2. Recovery of intersection grammar
3. Weights (intersection should allow lowest-weight derivations only)
4. MCFGs

5.1 Intersection with an FSM

To modify the algorithm to intersect a grammar with an FSM, we replace the input string with an FSM, and change our definition of a chart entry. Now, a chart entry is a 3-tuple $(i, C \rightarrow \alpha \bullet \beta, j)$, where i represents the first FSM state covered by α and j represents the last FSM state covered. We define an FSM here as a 5-tuple $\langle Q, \Sigma, s, F, M \rangle$, where Q is the set of states in the FSM, Σ is the label alphabet for the FSM (for our purposes Σ is always the same as V for all grammars in use), $s \in Q$ is the start state, $F \subseteq Q$ is the set of final states of the FSM, and M is a set of 3-tuples (i, a, j), which represent transitions from state i to state j with label a. Given these redefinitions we can then just modify the scan rule:

scan: $\frac{(i,C\rightarrow\alpha\bullet a\beta,j)}{(i,C\rightarrow\alpha a\bullet\beta,j+1)}$ if $(j, a, k) \in M$, where M is the input FSM.

and the predict rule in the obvious way:

predict: $\frac{(i,C\rightarrow\alpha\bullet A\beta,j)}{(j,A\rightarrow\bullet\gamma,j)}$ if $A \rightarrow \gamma \in P$ (if γ is of the form $a\ \gamma'$, $(j, a, k) \in M$ must hold as well)

Note that the initial entry in the chart is now $(s, S \rightarrow \bullet\alpha, s)$.

5.2 Grammar Recovery

It is possible to recover the output of intersection by increasing slightly what is in the chart. In particular, for every item on the chart, we note how it got there (just the last step). Each item on the chart may be referred to by its column number C and its position N within that column. We annotate only items produced by scan and complete steps, as follows:

- sC/N
- $cC_1/N_1; C_2/N_2$

where C_1/N_1 refers to the $(j, A \rightarrow \gamma\bullet, k)$ item from the complete step, and C_2/N_2 refers to the

$$
\begin{array}{rcl}
S & \rightarrow & \textit{Non Rd Non} \\
 & | & \textit{Rd Non} \\
 & | & \textit{Non Rd} \\
 & | & \textit{Rd} \\
\textit{Non} & \rightarrow & \textit{SSR UR MRD NBR} \\
 & | & \textit{Non SSR UR MRD NBR} \\
\textit{SSR} & \rightarrow & A\ A \\
\textit{UR} & \rightarrow & A\ A \\
\textit{MRD} & \rightarrow & -\ - \\
\textit{Rd} & \rightarrow & \textit{Rd1}_0\ \textit{Rd1}_1 \\
\textit{BDR} & \rightarrow & \left(\begin{array}{l} \textit{BDR0}_0\ \textit{BDR1}_0, \\ \textit{BDR0}_1\ \textit{BDR1}_1 \end{array} \right) \\
\textit{BDL} & \rightarrow & \left(\begin{array}{l} \textit{BDL0}_0\ \textit{BDL1}_0, \\ \textit{BDL0}_1\ \textit{BDL1}_1 \end{array} \right) \\
B & \rightarrow & \left(\begin{array}{l} \textit{B0}_0\ \textit{B1}_0, \\ \textit{B0}_1\ \textit{B1}_1 \end{array} \right) \\
A & \rightarrow & - \mid + \mid [\mid] \mid | \\
\textit{BDR}_n & \rightarrow & \left(\begin{array}{c|c|c|c|c} -, & +, & [, &], & |, \\ - & + & [& - & [\end{array} \right) \\
\textit{BDL}_n & \rightarrow & \left(\begin{array}{c|c|c|c|c} -, & +, & [, &], & |, \\ - & + & - &] &] \end{array} \right) \\
B_n & \rightarrow & \left(\begin{array}{c|c|c|c|c} -, & +, & [, &], & |, \\ - & + & [&] & | \end{array} \right)
\end{array}
$$

continuing with

$$
\begin{array}{rcllllll}
\textit{NBR} & \rightarrow & A & A & - & - & - & \\
\textit{RLE} & \rightarrow & A & A & - & [& - & \\
\textit{RRE} & \rightarrow & A & A & - &] & - & \\
\textit{BLE} & \rightarrow & A & A & A & - & [\\
\textit{BRE} & \rightarrow & A & A & A & - &] \\
\textit{RB} & \rightarrow & A & A & A &] & [\\
\textit{BR} & \rightarrow & A & A & A & [&] \\
\textit{RED} & \rightarrow & A & A & - & + & - & \\
\textit{BAS} & \rightarrow & A & A & A & - & +
\end{array}
$$

In cases where the reduplicant precedes the base, the reduplication rules will appear as follows:

$$
\begin{array}{rcl}
\textit{Rd1} & \rightarrow & \left(\begin{array}{lllll} \textit{BDR}_0 & \textit{UR} & \textit{MRD} & \textit{RLE} & \textit{Rd2}_0, \\ \textit{BDL}_0 & \textit{UR} & \textit{BDR}_1 & \textit{RB} & \textit{Rd2}_1 \\ \textit{SSR} & \textit{UR} & \textit{BDL}_1 & \textit{BRE} & \end{array} \right) \\
\textit{Rd2} & \rightarrow & \left(\begin{array}{llll} B_0 & \textit{UR} & \textit{MRD} & \textit{RED}, \\ \textit{SSR} & \textit{UR} & B_1 & \textit{BAS} \end{array} \right) \\
 & | & \left(\begin{array}{lllll} \textit{Rd2}_0 & B_0 & \textit{UR} & \textit{MRD} & \textit{RED}, \\ \textit{Rd2}_1 & \textit{SSR} & \textit{UR} & B_1 & \textit{BAS} \end{array} \right)
\end{array}
$$

Otherwise, where the base precedes the reduplicant, the rules will appear as follows:

$$
\begin{array}{rcl}
\textit{Rd1} & \rightarrow & \left(\begin{array}{lllll} \textit{SSR} & \textit{UR} & \textit{BDR}_1 & \textit{BLE} & \textit{Rd2}_0, \\ \textit{BDR}_0 & \textit{UR} & \textit{BDL}_1 & \textit{BR} & \textit{Rd2}_1 \\ \textit{BDL}_0 & \textit{UR} & \textit{MRD} & \textit{RRE} & \end{array} \right) \\
\textit{Rd2} & \rightarrow & \left(\begin{array}{llll} \textit{SSR} & \textit{UR} & B_1 & \textit{BAS}, \\ B_0 & \textit{UR} & \textit{MRD} & \textit{RED} \end{array} \right) \\
 & | & \left(\begin{array}{lllll} \textit{Rd2}_0 & \textit{SSR} & \textit{UR} & B_1 & \textit{BAS}, \\ \textit{Rd2}_1 & B_0 & \textit{UR} & \textit{MRD} & \textit{RED} \end{array} \right)
\end{array}
$$

Figure 4: Reduplication Grammar

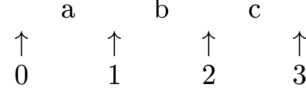


Figure 5: Numbering of string positions in the string "abc"

$(i, C \rightarrow \alpha \bullet A\beta, j)$ item. A chart item is thus now a 4-tuple $(i, C \rightarrow \alpha \bullet \beta, j, H)$, where H is a set of history items of the type described here, one for each scan or complete step that put the item there.

Recovery of a grammar then starts from the "success items," that is items in the chart that begin in state 1 and end with a final state and represent a production from the start symbol of the grammar, with the Earley position dot at the end of the production. We then move from right to left within those productions, filling in the state pairs for each constituent we pass, and tracing through their productions as well. Whenever we get to the left side of a production, we output it. The exact algorithm is as follows:

GrammarRecovery(*chart*)

> *queue* ← []
> **for all** *success items* $(s, S \rightarrow \gamma\bullet, f \in F, H_0)$ at (C, N) **do**
>   queue up (C, N) onto *queue*
>   **while** *queue* not empty **do**
> (C, N) ← dequeue from *queue*
>   *item* ← item at (C, N): $(i, A \rightarrow \alpha\bullet, j, H_1)$
>   *pos* ← pos. of • in *item*
>   *RHSs* ← **GetRHSs**([[]], *item, pos, queue*)
>   **for all** *RHS* ∈ *RHSs* **do**
>   output "$A(i, j) \rightarrow$ *RHS*"
>   **end for**
>   **end while**
> **end for**

GetRHSs(*rhss, item, pos, queue*)

> **if** *pos* $= 0$ **then**
>   return *rhss*
> **end if**
> *new_rhss* ← []
> **for all** history path components *hitem* of *item* **do**
>   *rhss*′ ← copy *rhss*
>   **extend**(*rhss*′, *hitem, pos, queue*)

add $rhss'$ to *new_rhss*
end for
return *new_rhss*

extend(*rhss, hitem, pos, queue*)

if $hitem = \mathrm{s}(C, N)$ **then**
prepend scanned symbol to each *rhs* $\in$ *rhss*
$prev \leftarrow$ item at (C, N)
else if $hitem = \mathrm{c}(C_1/N_1; C_2/N_2)$ **then**
$(i, A \rightarrow \gamma\bullet, j, H) \leftarrow$ item at (C_1, N_1)
prepend $A(i, j)$ to each $rhs \in rhss$
enter (C_1, N_1) into *queue*
$prev \leftarrow$ item at (C_2, N_2)
end if
return **GetRHSs(***rhss, item, pos*-1, *queue***)**

5.3 Weights

The basic idea for handling weights is an adaption from the Viterbi algorithm, as used for chart parsing of probabilistic grammars. Basically, we reduce the grammar to allow only the lowest-weight derivations from each new category.

Implementation: Each chart item has an associated weight, computed as follows:

predict: weight of the predicted rule $A \rightarrow \gamma$

scan: sum of the weight of the item scanned from and the weight of the FSM edge scanned across.

complete: sum of the weights of the two items involved

We build new chart items whenever permitted by the rules given in previous sections, assigning weights to them by the above considerations. If no equivalent item (equivalence ignores weight and path to the item) is in the chart, we add the item. If an equivalent item is in the chart, there are three possible actions, according to the weight of the new item:

1. Higher than the old item: do nothing (don't add the new path).
2. Lower than the old item: remove all other paths to the item, add this path to the item. Adjust weights of all items built from this one downward.
3. Same as the old item: add the new path to the item.

A chart item is thus now a 5-tuple $(w, i, C \rightarrow \alpha \bullet \beta, j, H)$, where w represents a weight, and all the other items are as before.

5.4 MCFGs

To extend the Earley algorithm to MCFGs, we first reduce the chart-building part of the Earley algorithm for MCFGs to the already-worked out algorithm for CFGs by converting the MCFG into a (not-equivalent) CFG. We then modify the grammar-recovery step to convert the CFG produced into an MCFG, verifying that the MCFG produced is a proper one.

5.4.1 Adjustments to the Chart-Building Algorithm:

First, we treat each part of the rule as a separate rule, and use the regular algorithm. Thus, $\mathrm{B} \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ becomes $\mathrm{B}_0 \rightarrow 0$ and $\mathrm{B}_1 \rightarrow 1$. Having separated a single rule such as $C \rightarrow (\alpha, \beta)$ into two parts $C_0 \rightarrow \alpha$ and $C_1 \rightarrow \beta$, we need to keep track, when building the chart and after, of which rule in the associated MCFG each chart item refers to. These annotations will be useful in Grammar Recovery (something like $C_0 \rightarrow \alpha$ can only be combined with $C_1 \rightarrow \beta$ if they both come from the same MCFG rule). Thus, a chart item is a 6-tuple $(r, w, i, C \rightarrow \alpha \bullet \beta, j, H)$, where r is the rule number from the original MCFG to which the production $C \rightarrow \alpha \bullet \beta$ corresponds, and all the others are as before.

5.4.2 Adjustments to Grammar Recovery

As before, followed by a final combinatory and checking step:

for all non-terminals A with arity n **do**
for all possible combinations $A_0(i, j) \rightarrow \gamma_0, A_1(k, l) \rightarrow \gamma_1, \ldots, A_n(m, n) \rightarrow \gamma_n$ **do**
if the MCFG condition applies to the combination **then**
output $A(i, j)(k, l) \ldots (m, n) \rightarrow (\gamma_0, \gamma_1, \ldots, \gamma_n)$
end if
end for
end for

where the MCFG condition is as follows:

All γ_i on the right hand side of the combination must be derived from the

same rule in the original set of rules and their yields must not overlap each other in the FSM.

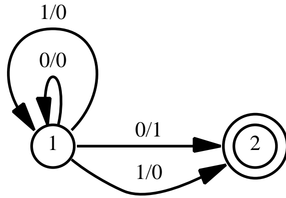
Given the way the chart-parsing and recovery algorithms work, the MCFG condition will be satisfied if we simply check that all the elements of the combination come from the same rule in the original MCFG. This will result in some invalid rules in the output grammar, but this simple check guarantees that these rules will be such that they will be unable to participate in derivations, since their right-hand sides will refer to categories that do not head any productions.

5.5 Example

As an example, let's take the simple reduplication grammar from before:

$$
\begin{array}{llll}
(1) & S & \rightarrow & A_0\,A_1 \\
(2) & A & \rightarrow & (1,1) \\
(3) & & | & (0,0) \\
(4) & & | & (0\,A_0, 0\,A_1) \\
(5) & & | & (1\,A_0, 1\,A_1)
\end{array}
$$

and intersect it with the machine



This machine generates the set of strings $\{0|1\}^+$, but weights all strings ending with 0.

The corresponding CFG-grammar used for the chart-building step is as follows:

$$
\begin{array}{llll}
(1) & S & \rightarrow & A_0\,A_1 \\
(2) & A_0 & \rightarrow & 1 \\
 & A_1 & \rightarrow & 1 \\
(3) & A_0 & \rightarrow & 0 \\
 & A_1 & \rightarrow & 0 \\
(4) & A_0 & \rightarrow & 0\,A_0 \\
 & A_1 & \rightarrow & 0\,A_1 \\
(5) & A_0 & \rightarrow & 1\,A_0 \\
 & A_1 & \rightarrow & 1\,A_1
\end{array}
$$

The chart produced by the chart-building part of the algorithm is as follows:

Column 1 ($j = 1$, $i = 1$)

#	r	w	$\in P$	H
0	1	0	$S \rightarrow \bullet A_0\,A_1$	$\emptyset$
1	2	0	$A_0 \rightarrow \bullet 1$	$\emptyset$
2	3	0	$A_0 \rightarrow \bullet 0$	$\emptyset$
3	4	0	$A_0 \rightarrow \bullet 0\,A_0$	$\emptyset$
4	5	0	$A_0 \rightarrow \bullet 1\,A_0$	$\emptyset$
5	5	0	$A_0 \rightarrow 1 \bullet A_0$	$\{s1/4\}$
6	4	0	$A_0 \rightarrow 0 \bullet A_0$	$\{s1/3\}$
7	3	0	$A_0 \rightarrow 0\bullet$	$\{s1/2\}$
8	1	0	$S \rightarrow A_0 \bullet A_1$	$\{c1/7; 1/0, c1/10; 1/0, c1/9; 1/0, c1/22; 1/0\}$
9	5	0	$A_0 \rightarrow 1\,A_0\bullet$	$\{c1/7; 1/5, c1/10; 1/5, c1/9; 1/5, c1/22; 1/5\}$
10	4	0	$A_0 \rightarrow 0\,A_0\bullet$	$\{c1/7; 1/6, c1/10; 1/6, c1/9; 1/6, c1/22; 1/6\}$
11	2	0	$A_1 \rightarrow \bullet 1$	$\emptyset$
12	3	0	$A_1 \rightarrow \bullet 0$	$\emptyset$
13	4	0	$A_1 \rightarrow \bullet 0\,A_1$	$\emptyset$
14	5	0	$A_1 \rightarrow \bullet 1\,A_1$	$\emptyset$
15	5	0	$A_1 \rightarrow 1 \bullet A_1$	$\{s1/14\}$
16	4	0	$A_1 \rightarrow 0 \bullet A_1$	$\{s1/13\}$
17	3	0	$A_1 \rightarrow 0\bullet$	$\{s1/12\}$
18	1	0	$S \rightarrow A_0\,A_1\bullet$	$\{c1/17; 1/8, c1/20; 1/8, c1/19; 1/8, c1/21; 1/8\}$
19	5	0	$A_1 \rightarrow 1\,A_1\bullet$	$\{c1/17; 1/14, c1/20; 1/14, c1/19; 1/14, c1/21; 1/14\}$
20	4	0	$A_1 \rightarrow 0\,A_1\bullet$	$\{c1/17; 1/13, c1/20; 1/13, c1/19; 1/13, c1/21; 1/13\}$
21	2	0	$A_1 \rightarrow 1\bullet$	$\{s1/11\}$
22	2	0	$A_0 \rightarrow 1\bullet$	$\{s1/1\}$

Column 2 ($j = 2$, $i = 1$)

#	r	w	$\in P$	H
0	5	0	$A_0 \rightarrow 1 \bullet A_0$	$\{s1/4\}$
1	4	1	$A_0 \rightarrow 0 \bullet A_0$	$\{s1/3\}$
2	3	1	$A_0 \rightarrow 0\bullet$	$\{s1/2\}$
3	1	0	$S \rightarrow A_0 \bullet A_1$	$\{c2/13; 1/0, c2/5; 1/0, c2/4; 1/0\}$
4	5	0	$A_0 \rightarrow 1\,A_0\bullet$	$\{c2/13; 1/5, c2/5; 1/5, c2/4; 1/5\}$
5	4	0	$A_0 \rightarrow 0\,A_0\bullet$	$\{c2/13; 1/6, c2/5; 1/6, c2/4; 1/6\}$
6	5	0	$A_1 \rightarrow 1 \bullet A_1$	$\{s1/14\}$
7	4	1	$A_1 \rightarrow 0 \bullet A_1$	$\{s1/13\}$
8	3	1	$A_1 \rightarrow 0\bullet$	$\{s1/12\}$
9	1	0	$S \rightarrow A_0\,A_1\bullet$	$\{c2/12; 1/8, c2/11; 1/8, c2/10; 1/8\}$
10	5	0	$A_1 \rightarrow 1\,A_1\bullet$	$\{c2/12; 1/15, c2/11; 1/15, c2/10; 1/15\}$
11	4	0	$A_1 \rightarrow 0\,A_1\bullet$	$\{c2/12; 1/16, c2/11; 1/16, c2/10; 1/16\}$
12	2	0	$A_1 \rightarrow 1\bullet$	$\{s1/11\}$
13	2	0	$A_0 \rightarrow 1\bullet$	$\{s1/1\}$

In this chart the items with an empty history list were entered by prediction steps. The "success item" for this grammar is then item (2,9): $(r = 1, w = 0, i = 1, p = S \rightarrow A_0\,A_1\bullet, j =$

$2, H = \{c2/12; 1/8, c2/11; 1/8, c2/10; 1/8\})$, so begin there:

$$S(1,2) \rightarrow A_0\, A_1$$

We then queue up (2,12), (2,11), and (2,10), noting that for all of these the states for A_1 are (1,2), and we move to item (1,8): $(r = 1, w = 0, i = 1, p = S \rightarrow A_0 \bullet A_1, j = 1, H = \{c1/7; 1/0, c1/10; 1/0, c1/9; 1/0, c1/22; 1/0\})$. Here we queue up (1,7), (1,10), (1,9), and (1,22), noting that for all of these the states for A_0 are (1,1). Moving to (1,0), we note that we are done, and we thus output a complete rule:

$$(r1)\ S(1,2) \rightarrow A_0(1,1)\, A_1(1,2).$$

We then encounter (2,12) on the queue: $(r = 2, w = 0, i = 1, p = A_1 \rightarrow 1\bullet, j = 2, H = \{s1/11\})$, which can be output with no further ado:

$$(r2)\ A_1(1,2) \rightarrow 1$$

Moving to item (2,11) $(r = 4, w = 0, i = 1, p = A_1 \rightarrow 0\ A_1\bullet, j = 2, H = \{c2/12; 1/16, c2/11; 1/16, c2/10; 1/16\})$ we don't need to queue anything, and we can see that the output will be:

$$(r4)\ A_1(1,2) \rightarrow 0\ A_1(1,2)$$

Item (2,10) is $(r = 5, w = 0, i = 1, p = A_1 \rightarrow 1\ A_1\bullet, j = 2, H = \{c2/12; 1/15, c2/11; 1/15, c2/10; 1/15\})$, so we output

$$(r5)\ A_1(1,2) \rightarrow 1\ A_1(1,2)$$

We now move on to item (1,7): $(r = 3, w = 0, i = 1, p = A_0 \rightarrow 0\bullet, j = 1, H = \{s1/2\})$, which we output as

$$(r3)\ A_0(1,1) \rightarrow 0.$$

Item (1,10) is $(r = 4, w = 0, i = 1, p = A_0 \rightarrow 0\ A_0\bullet, j = 1, H = \{c1/7; 1/6, c1/10; 1/6, c1/9; 1/6, c1/22; 1/6\})$. In dealing with this we need to queue nothing, and we output:

$$(r4)\ A_0(1,1) \rightarrow 0\ A_0(1,1)$$

Moving to (1,9), which is $(r = 5, w = 0, i = 1, p = A_0 \rightarrow 1\ A_0\bullet, j = 1, H = \{c1/7; 1/5, c1/10; 1/5, c1/9; 1/5, c1/22; 1/5\})$, we queue nothing and output

$$(r5)\ A_0(1,1) \rightarrow 1\ A_0(1,1)$$

Finally we get to (1,22): $(r = 2, w = 0, i = 1, p = A_0 \rightarrow 1\bullet, j = 1, H = \{s1/1\})$, which gets output as

$$(r2)\ A_0(1,1) \rightarrow 1$$

Collecting these together (for category A), we get the following pairings:

$$\left(\begin{array}{c|lcl|lcl}
(r2) & A_0(1,1) & \rightarrow & 1 & A_1(1,2) & \rightarrow & 1 \\
(r3) & A_0(1,1) & \rightarrow & 0 & & & \\
(r4) & A_0(1,1) & \rightarrow & 0\, A_0(1,1) & A_1(1,2) & \rightarrow & 0\, A_1(1,2) \\
(r5) & A_0(1,1) & \rightarrow & 1\, A_0(1,1) & A_1(1,2) & \rightarrow & 1\, A_1(1,2)
\end{array}\right)$$

Note that the "pair" for (r3) has no second member, so nothing will be output for it. Combining the compatible rules, we get the following grammar:

$$\begin{array}{rcl}
S(1,2) & \rightarrow & A(1,1)(1,2)_0\, A(1,1)(1,2)_1 \\
A(1,1)(1,2) & \rightarrow & (1,1) \\
& | & (0\, A_0(1,1)(1,2), 0\, A_1(1,1)(1,2)) \\
& | & (1\, A_0(1,1)(1,2), 1\, A_1(1,1)(1,2))
\end{array}$$

which is equivalent to the grammar:

$$\begin{array}{rcl}
S & \rightarrow & A_0\, A_1 \\
A & \rightarrow & (1,1) \\
& | & (0\, A_0, 0\, A_1) \\
& | & (1\, A_0, 1\, A_1)
\end{array}$$

This grammar indeed represents the best outputs from the intersection—all reduplicating forms which end in a 1.

References

Daniel M. Albro. 1998. Evaluation, implementation, and extension of Primitive Optimality Theory. Master's thesis, UCLA.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Comm. of the ACM*, 6(2):451–455.

Jason Eisner. 1997a. Efficient generation in primitive Optimality Theory. In *Proceedings of the ACL*.

Jason Eisner. 1997b. What constraints should OT allow? Handout for talk at LSA, Chicago, January.

John McCarthy and Alan Prince. 1995. Faithfulness and reduplicative identity. In J. Beckman, S. Urbanczyk, and L. Walsh, editors, *Papers in Optimality Theory*, number 18 in University of Massachusetts Occasional Papers, pages 259–384. GLSA, UMass, Amherst.

Alan Prince and Paul Smolensky. 1993. Optimality Theory: Constraint interaction in generative grammar. Technical Report 2, Center for Cognitive Science, Rutgers University.

H. Seki, T. Matsumura, M. Fujii, and T. Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Bruce Tesar. 1996. Computing optimal descriptions for optimality theory grammars with context-free position structures. In *Proceedings of ACL*.

Nikè van Vugt. 1996. Generalized context-free grammars. Master's thesis, Universiteit Leiden. Internal Report 96-12.

Markus Walther. 2000. Finite-state reduplication in one-level prosodic morphology. In *Proceedings of NAACL-2000*, pages 296–302, Seattle, WA.