A Finite-State Approximation of Optimality Theory: the Case of Finnish Prosody

Lauri Karttunen

Palo Alto Research Center, 3333 Coyote Hill Rd, Palo Alto CA 94304, USA
karttunen@parc.com
http://www2.parc.com/istl/members/karttune/

Abstract. This paper gives a finite-state formulation of two closely related descriptions of Finnish prosody proposed by Paul Kiparsky and Nine Elenbaas in the framework of OPTIMALITY THEORY. In native Finnish words, the primary stress falls on the first syllable. Secondary stress generally falls on every second syllable. However, secondary stress skips a light syllable that is followed by a heavy syllable. Kiparsky and Elenbaas attempt to show that the ternary pattern arises from the interaction of universal metrical constraints.
This paper formalizes the Kiparsky and Elenbaas analyses using the PARC/XRCE regular expression calculus. It shows how output forms with syllabification, stress and metrical feet are constructed from unmarked input forms. The optimality constraints proposed by Kiparsky and Elenbaas are reformulated in finite-state terms using lenient composition. The formalization shows that both analyses fail for many types of words.

1 Introduction

This article is a companion piece to Karttunen [1] that refutes the account proposed by Paul Kiparsky [2] and Nine Elenbaas [3, 4] in the framework of OPTIMALITY THEORY [5–7] for ternary rhythm in Finnish. The purpose of this follow-up article is to fill in the missing technical details and provide a complete account of the finite-state implementation that was used to derive the result.

In general, Finnish prosody is trochaic with the main stress on the first syllable and a secondary stress on every other following syllable. Finnish also has a ternary stress pattern that surfaces in words where the stress would fall on a light syllable that is followed by a heavy syllable. A light syllable ends with a short vowel (*ta*); a heavy syllable ends with a coda consonant (*jat*, *an*) or a long vowel (*kuu*, *aa*) or a diphthong (*voi*, *ei*). Example (1a) shows the usual trochaic pattern; (1b) starts off with a ternary foot.[1]

(1) a. (rá.kas).(tà.ja).(tàr.ta) 'mistress' (Sg. Par.)
    b. (rá.kas).ta.(jàt.ta).(rè.na) 'mistress' (Sg. Ess.)

[1] Kiparsky and Elenbaas treat the third syllable of a dactyl as extrametrical, that is, (rá.kas).ta. instead of (rá.kas.ta). This decision of not recognizing a ternary foot as a primitive is of no consequence as far as the topic of this paper is concerned.

The acute accent indicates primary stress and the grave accents mark secondary stress. Periods mark syllable boundaries and feet are enclosed in parentheses.

The fundamental idea in the Kiparsky and Elenbaas studies is that the ternary rhythm in examples such as (1b) arises naturally from the interaction of constraints that in other types of words produce a binary stress pattern. The fundamental assumption in the OT framework is that all types of prosodic structures are always available in principle. It is the constraints and the ranking between them that determines which of the competing candidates emerges as the winner. In some circumstances the constraints select the binary rhythm, in other circumstances the trochaic pattern wins. Unfortunately the constraints they propose pick out the wrong pattern in many cases. This fact has not been noticed by the proponents of the OT analyses. It is practically impossible to detect such errors without a computational implementation.

2 OT Constraints for Finnish Prosody

Under Kiparsky's analysis (p. 111), the prosody of Finnish is characterized by the system in (2). The constraints are listed in the order of their priority.

(2) a. *CLASH: No stresses on adjacent syllables.
b. LEFT-HANDEDNESS: The stressed syllable is initial in the foot.
c. MAIN STRESS: The primary stress in Finnish is on the first syllable.
d. FOOTBIN: Feet are minimally bimoraic and maximally disyllabic.
e. *LAPSE: Every unstressed syllable must be adjacent to a stressed syllable or to the word edge.
f. NON-FINAL: The final syllable is not stressed.
g. STRESS-TO-WEIGHT: Stressed syllables are heavy.
h. LICENSE-σ: Syllables are parsed into feet.
i. ALL-FT-LEFT: The left edge of every foot coincides with the left edge of some prosodic word.

Elenbaas [3] and Elenbaas and Kager [4] give essentially the same analysis except that they replace Kiparsky's STRESS-TO-WEIGHT constraint with the more specific one in (3).

(3) *(L̀H): If the second syllable of a foot is heavy, the stressed syllable should not be light.

3 Finite-State Approximation of OT

As we will see shortly, classical OT constraints such as those in (2) and (3) are REGULAR (= RATIONAL) in power. They can be implemented by finite-state networks. Nevertheless, it has been known for a long time (Frank and Satta [8], Karttunen [9], Eisner [10]) that OT as a whole is not a finite-state system. Although the official OT rhetoric suggests otherwise, OT is fundamentally more complex than finite-state models of phonology such as classical Chomsky-Halle phonology [11] and Koskenniemi's two-level model [12]. The reason is that OT takes into account not just the ranking of the constraints but the number of constraint violations. For example, (4a) and (4b) win over (4c) because (4c) contains two violations of *LAPSE whereas (4a) and (4b) have no violations.[2]

(4) a. (ér.go).(nò.mi).a 'ergonomics' (Nom. Sg.)
    b. (ér.go).no.(mì.a)
    c. (ér.go).no.mi.a

Furthermore, for GRADIENT constraints such as ALL-FT-LEFT, it is not just the number of instances of non-compliance that counts but the SEVERITY of the offense. Candidates (4a) and (4b) both contain one foot that is not at the left edge of the word. But they are not equally optimal. In (4a) the foot not conforming to ALL-FT-LEFT, (nò.mi), is two syllables away from the left edge whereas in (4b) the noncompliant (mì.a) is three syllables away from the beginning. Consequently, (4b) with three violations of ALL-FT-LEFT loses to (4a) that only has two violations of that constraint.

If the number of constraint violations is bounded, the classical OT theory of [5] can be approximated by a finite-state cascade where the input is first composed with a transducer, GEN, that maps the input to a set of output candidates (possibly infinite) and the resulting input/output transducer is then "leniently" composed with constraint automata starting with the most highly ranked constraint. We will use this technique, first described in [9], to implement the two OT descriptions of Finnish prosody. The key operation, LENIENT COMPOSITION, is a combination of ordinary composition and PRIORITY UNION [13].

Priority Union The priority union operator `.P.` is defined in terms of other regular expression operators in the PARC/XRCE calculus.[3] The definition of priority union is given in (5).

(5) `Q .P. R` $=_{def}$ `Q | [~[Q.u] .o. R]`

The `.u` operator in (5) extracts the "upper" language from a regular relation; ~ is negation. Thus the expression `~[Q.u]` denotes the set of strings that do not occur on the upper side of the `Q` relation. The symbol `.o.` is the composition operator and `|` stands for union. The effect of the composition `[~[Q.u] .o. R]` is to restrict `R` to mappings of strings that are not mapped into anything in `Q`. Only this subrelation of `R` is unioned with `Q`. In other words, `[Q .P. R]` gives precedence to the mappings in `Q` over the mappings in `R`.

Lenient Composition The basic idea of lenient composition can be explained as follows. Assume that `R` is a relation, a mapping that assigns to each input form some number of outputs, and that `C` is a constraint that prohibits some of

[2] It is important to keep in mind that the actual scores, 0 vs. 2, are not relevant. What matters is that (4a) and (4b) have **fewer** violations than (4c).
[3] The PARC/XRCE regular expression formalism is presented in Chapter 2 of [14].

the output forms. The lenient composition of `R` and `C`, denoted as `[R .O. C]`, is the relation that eliminates all the output candidates of a given input that do not conform to `C`, provided that the input has at least one output that meets the constraint. If none of the output candidates of a given input meet the constraint, lenient composition spares all of them. Consequently, every input will have at least one output, no matter how many violations it incurs.[4]

We define the desired operation, denoted `.O.`, as a combination of ordinary composition and priority union in (6).

(6) `R .O. C` $=_{def}$ `[R .o. C] .P. R`

The left side of the priority union in (6), `[R .o. C]` restricts `R` to mappings that satisfy the constraint `C`. That is, any pair whose lower side string is not in `C` will be eliminated. If some string in the upper language of `R` has no counterpart on the lower side that meets the constraint, then it is not present in `[R .o. C].u` but, for that very reason, it will be "rescued" by the priority union. In other words, if an underlying form has some output that can meet the given constraint, lenient composition enforces the constraint. If an underlying form has no output candidates that meet the constraint, then the underlying form and all its outputs are retained. The definition of lenient composition entails that the upper language of `R` is preserved in `[R .O. C]`.

In order to be able to give preference to output forms that incur the fewest violations of a constraint `C`, we first mark the violations and then select the best candidates using lenient composition. We set a limit n, an upper bound for the number of violations that the system will consider, and employ a set of auxiliary constraints, $\mathtt{V}_{n-1}$, $\mathtt{V}_{n-2}$, ..., $\mathtt{V}_0$, where $\mathtt{V}_i$ accepts the output candidates that violate the constraint at most i times. The most stringent enforcer, $\mathtt{V}_0$, allows no violations. Given a relation `R`, a mapping from the inputs to the current set of output candidates, we mark all the violations of `C` and then prune the resulting `R'` with lenient composition: `R' .O.` $\mathtt{V}_{n-1}$ `.O.` $\mathtt{V}_{n-2}$ `... .O.` $\mathtt{V}_0$. If an input form has output candidates that are accepted by $\mathtt{V}_i$, where $n > i \geq 0$, all the ones that are rejected by $\mathtt{V}_i$ are eliminated; otherwise the set of output candidates is not reduced. The details of this strategy are explained in Section 4.2.

4 Finite-State OT Prosody

In this section, we will show how the two OT descriptions of Finnish prosody in Section 2 can be implemented in a finite-state system. The regular expression formalism in this section and the **xfst** application used for computation are described in the book *Finite State Morphology* [14].

The first objective is to provide a definition of the GEN function for Finnish prosody. The function must accomplish three tasks: (1) parse the input into syllables, (2) assign optional stress, and (3) combine syllables optionally into metrical feet. In keeping with the hallmark OT thesis of "freedom of analysis",

[4] Frank and Satta [8, pp. 8–9] call this operation "conditional intersection."

we need a prolific GEN. Every conceivable output candidate, however bizarre, should be made available for evaluation by the constraints.

The second objective is to express Kiparsky's nine constraints in (2) in finite-state terms and bundle them together in the order of their ranking. Combined with GEN, the system should map any input into its optimal metrical realization in Finnish.

4.1 The GEN function

To simplify our definitions of complex regular expressions, it is useful to start with some elementary notions and define higher-level concepts with the help of the finite-state calculus.

Basic Definitions Each of the definitions in (7) is a formula in the PARC/XRCE extended regular expression language and compiles into a finite-state network. The vertical bar, `|`, is the union operator. Consequently, the first statement in (7) defines `HighV` as the language consisting of the strings *u*, *y* and *i*. The text following `#` is a comment.

(7)
```
define HighV [u | y | i];                         # High vowel
define MidV [e | o | ö];                           # Mid vowel
define LowV [a | ä] ;                              # Low vowel
define USV [HighV | MidV | LowV];          # Unstressed Vowel
define C [b | c | d | f | g | h | j | k | l | m |
          n | p | q | r | s | t | v | w | x | z]; # Consonant

define MSV [á | é | í | ó| ú | ý | ä́ | ö́];      # Main stress
define SSV [à | è | ì | ò | ù | ỳ | ä̀ | ö̀];# Secondary stress
define SV [MSV | SSV];                        # Stressed vowel
define V [USV | SV] ;                                  # Vowel
define P [V | C];                                    # Phoneme
```

We also need some auxiliary symbols to mark syllable and foot boundaries. The auxiliary alphabet is defined in (8). The period, `.`, marks internal syllable boundaries. The parentheses, `( )`, enclose a metrical foot. We use a special symbol, `.#.`, to refer to the beginning or the end of a string. The suffix operator, `+`, creates a "one-or-more" iterative language from whatever it is attached to.

(8)
```
define B [["(" | ")" | "." ]+ | .#.];              # Boundary
define E .#. | ".";                                     # Edge
```

Two basic types of syllables are defined in (9). The onset of a syllable may contain zero or more consonants, `C*`. The nucleus of a light syllable, `LS`, consists of a single short vowel. For example, *a*, *ta* and *stra* are light syllables.

(9)
```
define LS [C* V];                             # Light Syllable
define HS [LS P+];                            # Heavy Syllable
define S [HS | LS];                                 # Syllable
```

In addition to knowing whether a syllable is light or heavy, we also need to know about the stress. The ampersand, `&`, in (10) stands for intersection and the dollar sign, `$`, is the “contains” operator. A stressed syllable, `SS`, is thus the intersection of all syllables with anything that contains a stressed vowel. With ∼ standing for negation, an unstressed syllable, `US`, is the intersection of all syllables with anything that does not contain a stressed vowel. Finally, `MSS` is a syllable with a vowel that has the main stress.

```
(10)  define SS [S & $SV];                        # Stressed Syllable
      define US [S & ∼$SV];                     # Unstressed Syllable
      define MSS [S & $MSV] ;              # Syllable with Main Stress
```

With the help of the basic concepts in (7)-(10) we can proceed to the first real task, the definition of Finnish syllabification.

Syllabification Assigning the correct syllable structure is a non-trivial task in Finnish because the nucleus of a syllable may consist of a short vowel, a long vowel, or a diphthong. A diphthong is a combination of two unlike vowels that together form the nucleus of a syllable. Adjacent vowels that cannot constitute a long vowel or a diphthong must be separated by a syllable boundary. In general, Finnish diphthongs end in a high vowel. However, in the first syllable there are three exceptional high-mid diphthongs: *ie*, *uo*, and *yö* that historically come from long *ee*, *oo*, and *öö*, respectively. All other adjacent vowels must be separated by a syllable boundary. For example, the first *ie* in the input *sienien* ‘mushroom’ (Pl. Gen.) constitutes a diphthong but the second *ie* does not because it is not in the first syllable. The correct syllabification is *sie.ni.en*.[5]

We will define `Syllabification` as a transducer that takes any input and inserts periods to mark syllable boundaries. Because of the issue with diphthongs, it is convenient to build the final syllabification transducer from two components. The first one, `MarkNonDiphth`, in (11) inserts syllable boundaries (periods) between vowels that cannot form the nucleus of a syllable.

```
(11)  define MarkNonDiphth [ [. .] -> "." ||
                                   [HighV | MidV] _ LowV, # i.a, e.a
                                   LowV _ MidV,           # a.e
                                   i _ [MidV - e],        # i.o, i.ö
                                   u _ [MidV - o],        # u.e
                                   y _ [MidV - ö],        # y.e
                                   $V i _ e ];            # sieni.en
```

The arrow, `->` is the “replace” operator. The first line of (11) specifies that an epsilon (empty string), `[. .]`[6], is replaced by a period in certain contexts, defined on the six lines following `||`. The underscore, _, marks the site of the of the

[5] Instead of providing the syllabification directly as part of GEN, it would of course be possible to generate a set of possible syllabification candidates from which the winners would emerge through an interaction with OT constraints such as HaveOnset, FillNucleus, NoCoda, etc.

[6] For an explanation of the `[. .]` notation, see [14, pp. 67–68]

replacement between left and right contexts. For example, the last context line in (11) inserts a syllable boundary between i and e when there is some preceding vowel. Thus it breaks the second, but not the first, *ie*-cluster in in words such as *sienien* 'mushroom' (Pl. Gen.). The minus symbol, `-`, in the preceding three lines denotes subtraction, e.g. `[MidV - e]` is any mid vowel other than *e*.

The second component of syllabification is defined in (12). Here `@->` is the left-to-right, longest-match replace operator. The effect of the rule is to insert a syllable boundary after a maximal match for the `C* V+ C*` pattern provided that it is followed by a consonant and a vowel. Here ... mark the match for the pattern and "." is the insertion after the match. Applying `MaximizeSyll` to an input string such as *strukturalismi* yields *struk.tu.ra.lis.mi.*

```
(12)  define MaximizeSyll [ C* V+ C* @-> ... "." || _ C V ];
```

Having defined the two components separately, we can now define the general syllabification rule by composing them together into a single transducer, as shown in (13) where `.o.` is the ordinary composition operator.

```
(13)  define Syllabify [MarkNonDiphth .o. MaximizeSyll];
```

For example, when `Syllabify` is applied to the input *sienien*, the outcome is *sie.ni.en* where the second syllable boundary comes from `MarkNonDiphth` and the first one from `MaximizeSyll`.

The general syllabification rule in (13) has exceptions. In particular, some loan words such as *ate.isti* 'atheist' must be partially syllabified in the lexicon. Compound boundaries must be indicated to prevent bad syllabifications such as **i.soi.sä* for *i.so#i.sä* 'grand father'.

Stress Because the proper distribution of primary and secondary stress is determined by the optimality constraints, all that the GEN function needs to do is to allow any vowel to have a main stress, a secondary stress or be unstressed. This is accomplished by the definition in (14) where `(->)` is the "optional replace" operator. The effect of the rule is to optionally replace each of the six vowels by the two stressed versions of the same vowel. For example, `OptStress` maps the input *maa* into *maa*, *máa* and *màa*.

```
(14)  define OptStress [ a (->) á|à, e (->) é|è, i (->) í|ì,
                         o (->) ó|ò, u (->) ú|ù, y (->) ý|ỳ,
                         ä (->) ä́|ä̀, ö (->) ö́|ö̀  ||  E C* _ ];
```

Because of the context restriction, `E C* _`, in (14), the stress is always assigned to the first component of a long vowel or a diphthong. As defined in (8), `E` stands here for a syllable boundary or the beginning of a word.

Metrical Structure In keeping with the OT philosophy, the grouping of syllables into metrical feet should also be done optionally and in every possible way to create a rich candidate set for the evaluation by optimality constraints.

The definion of `OptScan` in (15) yields a foot-building transducer that optionally wraps parentheses around one, two or three adjacent syllables. The expression to the left of the optional replace operator, `[S ("." S ("." S)) & $SS]`, defines a pattern that matches one or two or three syllables with their syllable boundary marks. The intersection with `$SS` guarantees that at least one of them is a stressed syllable. The right side of the rule wraps any instance of such a pattern within parentheses thus creating a metrical foot. The context restriction, `E _ E`, has the effect that feet consist of whole syllables with no part left behind.

(15)
```
define OptScan [[S ("." S ("." S)) & $SS] (->) "(" ... ")"
                 || E _ E];
```

Assembling the GEN Function Having defined separately the three components of GEN, syllabification, stress assignment and footing, we can now build the GEN function by composing the three transducers with the definition in (16).

(16)
```
define GEN(X) [ X .o. Syllabify .o. OptStress .o. OptScan ];
```

where `X` can be a single input form or a symbol representing a set of input forms or an entire language. The result of compiling a regular expression of the form `GEN(X)` is a transducer that maps each input form in `X` into all of its possible output forms.

Because stress assignment and footing are optional, The `GEN()` function produces a large number of alternative prosodic structures for even short inputs. For example, for the input *kala* ‘fish’ (Sg. Nom.), `GEN({kala})` produces the 33 output forms shown in (17).

(17) kà.là, kà.lá, kà.la, kà.(lá), kà.(là), ká.là, ká.lá, ká.la, ká.(lá), ká.(là), ka.là, ka.lá, ka.la, ka.(lá), ka.(là), (ká).là, (ká).lá, (ká).la, (ká).(lá), (ká).(là), **(ká.la)**, (ká.lá), (ká.là), (kà).là, (kà).lá, (kà).la, (kà).(lá), (kà).(là), (kà.la), (kà.lá), (kà.là), (ka.lá), (ka.là)

As the analyses by Elenbaas and Kiparsky predict, the correct output is (ká.la).

4.2 The Constraints

There are two types of violable OT constraints. For CATEGORICAL constraints, the penalty is the same no matter where the violation occurs. For GRADIENT constraints, the site of violation matters. For example, ALL-FEET-LEFT assigns to non-initial feet a penalty that increases with the distance from the beginning of the word.

Our general strategy is as follows. We first define an evaluation template for the two constraint types and then define the constraints themselves with the help of the templates. We use asterisks as violation marks and use lenient composition to select the output candidates with the fewest violation marks. Categorical constraints mark each violation with an asterisk. Gradient constraints mark violations with sequences of asterisks starting from one and increasing with the distance from the word edge.

The initial set of output candidates is obtained by composing the input with GEN. As the constraints are evaluated in the order of their ranking, the number of output forms is successively reduced. At the end of the evaluation, each input form typically should have just one correct output form.

An evaluation template for categorical constraints, shown in (18), needs four arguments: the current output mapping, a regular expression pattern describing what counts as a violation, a left context, and a right context.[7]

(18)
```
define Cat(Candidates, Violation, Left, Right) [
   Candidates .o. Violation -> ... "*" || Left _ Right
   .O. Viol3 .O. Viol2 .O. Viol1 .O. Viol0
   .o. Pardon ];
```

The first part of the definition composes the candidate set with a rule transducer that inserts an asterisk whenever it sees a violation that occurs in the specified context. The second part of the definition is a sequence of lenient compositions. The first one eliminates all candidates with more than three violations, provided that some candidates have only three or fewer violations. Finally, we try to eliminate all candidates with even one violation. This will succeed only if there are some output strings with no asterisks. The auxiliary terms `Viol3`, `Viol2`, `Viol1`, `Viol0` limit the number of asterisks. For example, `Viol1`, is defined as `~[$"*"]^2`. It prohibits having two or more violation marks. The third part, `Pardon`, is defined as `"*" -> 0`. It removes any remaining violation marks from the output strings. Because we are counting violations only up to three, we cannot distinguish strings that have four violations from strings with more than four violations. It turns out that three is an empirically sufficient limit for our categorical prosody constraints.

The evaluation template for gradient constraints counts up to 14 violations and each violation incurs more and more asterisks as we count instances of the left context. The definition is given in (19).

(19)
```
define GradLeft(Candidates, Violation, Left, Right) [
   Candidates
   .o. Violation -> "*" ... ||.#. Left _ Right
   .o. Violation -> "*"^2 ... ||.#. Left^2 _ Right
   .o. Violation -> "*"^3 ... ||.#. Left^3 _ Right
   .o. Violation -> "*"^4 ... ||.#. Left^4 _ Right
   .o. Violation -> "*"^5 ... ||.#. Left^5 _ Right
   .o. Violation -> "*"^6 ... ||.#. Left^6 _ Right
   .o. Violation -> "*"^7 ... ||.#. Left^7 _ Right
   .o. Violation -> "*"^8 ... ||.#. Left^8 _ Right
   .o. Violation -> "*"^9 ... ||.#. Left^9 _ Right
```

[7] Some constraints can be specified without referring to a particular left or right context. The expression `?*` stands for any unspecified context.

```
.o. Violation -> "*"^10 ... ||.#. Left^10 _ Right
.o. Violation -> "*"^11 ... ||.#. Left^11 _ Right
.o. Violation -> "*"^12... || .#. Left^12 _ Right
.o. Violation -> "*"^13 ... ||.#. Left^13 _ Right
.o. Violation -> "*"^14 ... ||.#. Left^14 _ Right
.O. Viol14 .O. Viol13 .O. Viol12 .O.Viol11 .O. Viol10
.O. Viol9 .O. Viol8 .O. Viol7 .O. Viol6 .O. Viol5
.O. Viol4 .O. Viol3 .O. Viol2 .O. Viol1 .O.  Viol0
.o. Pardon ];
```

Using the two templates in (18) and (19), we can now give very simple definitions for Kiparsky's nine constraints in (2).

(20) a. *Clash: No stress on adjacent syllables.
`define Clash(X) Cat(X, SS, SS B, ?*);`
b. Left-handedness: The stressed syllable is initial in the foot.
`define AlignLeft(X) Cat(X, SS, ".", ?*);`
c. Main Stress: The primary stress in Finnish is on the first syllable.
```
define MainStress(X)
  Cat(X, ~[B MSS ~$MSS], .#., .#.);
```
d. Foot-Bin: Feet are minimally bimoraic and maximally bisyllabic.
```
define FootBin(X)
  Cat(X, ["(" LS ")" | "(" S ["." S]^>1], ?*, ?*);
```
e. Lapse: Every unstressed syllable must be adjacent to a stressed syllable or to the word edge.
`define Lapse(X) Cat(X, US, [B US B], [B US B]);`
f. Non-Final: The final syllable is not stressed.
`define NonFinal(X) Cat(X, SS, ?*, ~$S .#.);`
g. Stress-To-Weight: Stressed syllables are heavy.
`define StressToWeight(X) Cat(X, [SS & LS], ?*, B);`
h. License-σ: Syllables are parsed into feet.
`define Parse(X) Cat(X, S, E, E);`
i. All-Ft-Left: The left edge of every foot coincides with the left edge of some prosodic word.
```
define AllFeetFirst(X)
  GradLeft(X, "(", [~$"." "." ~$"."], ?*);
```

To take just one example, let us consider the `StressToWeight` function. The violation part of the definition, `[SS & LS]`, picks out syllables such as *tí* and *tì* that are light and contain a stressed vowel. The left context is irrelevant, represented as `?*`. The right context matters. It must be some kind of boundary; otherwise perfectly well-formed outputs such as (má.te).ma.(tìik.ka) would get two violation marks: (má*.te).ma.(tì*ik.ka). That is because *tì* by itself is a stressed light syllable but *tìik* is not. The violation mark on the initial syllable *má* is correct but has no consequence because the higher-ranked `MainStress` constraint has removed all competing output candidates for *matematiikka* 'mathematics' (Sg. Nom.) that started with a secondary stress, *mà*, or without any stress, *ma*.

4.3 Combining GEN with the Constraints

Having defined both the GEN function and Kiparsky's nine prosody constraints, we can now put it all together creating a single function, `FinnishProsody`, that should map any Finnish input into its correct prosodic form. The definition is given in (21).

```
(21) define FinnishProsody(Input) [ AllFeetFirst( Parse(
   StressToWeight( NonFinal( Lapse( FootBin( MainStress( AlignLeft(
   Clash( GEN( Input )))))))))) ];
```

A regular expression of the form `FinnishProsody(X)` is computed "inside-out." First the GEN function defined in (16) maps each of the input forms in `X` into all of its possible output forms. Then the constraints defined in Section 4.2 are applied in the order of their ranking to eliminate violators, making sure that at least one output form remains for all the inputs. For example, the compilation of the regular expression in (22)

```
(22) FinnishProsody({rakastajatarta} | {rakastajattarena}) ;
```

produces a transducer with the mappings in (23) and (24).

```
(23)  r a   k a s      t a   j a      t a r   t a
    ( r á . k a s ) . ( t à . j a ) . ( t à r . t a )
```

```
(24)  r a   k a s     t a     j a t   t a      r e   n a
    ( r á . k a s ) . t a . ( j à t . t a ) . ( r è . n a )
```

This is the right result we already saw in (1). Unfortunately there are many input patters that yield an incorrect result. Some examples are given in (25). We use `L` for light, `H` for heavy syllable and `X` when the distinction between `L` and `H` does not matter. For a discussion of what goes wrong, see [1].

```
(25)     XXLLLX: *(ká.las).te.(lè.mi).nen
         XXHHLX: *(há.pa).roi.(tùt.ta).vaa
        XXLHHLX: *(pú.hu).(tè.tuim).(mìs.ta).kin
     XXHHLHHLX: *(jǟ́r.jes).tel.(mǟ̀l.li).syy.(dèl.lä).ni
```

Replacing Kiparsky's `StressToWeight` by Elenbaas' more specific *(L̀H) constraint helps in some cases and hurts in others. The last of the four patterns in (25) comes out correct but a new type of error appears, as shown in (26).

```
(26)     XXHLLX: *(kú.ti).tet.(tù.ja).kin
     XXHHLHHLX: (jǟ́r.jes).(tèl.mäl).li.(sỳ̈y.del).(lǟ̀.ni)
```

5 Conclusion

The basic assumption in the Kiparsky and Elenbaas & Kager studies is that the alternation between binary and ternary patterns in Finnish arises in a natural way from the interaction of universal constraints. It would be a satisfying result but, unfortunately, it is not true for the constraints that have been proposed so far. The traditional tableau method commonly used by phonologists cannot handle the vast number of competing output candidates that the theory postulates. Computational techniques such as those developed in this article are indispensable in finding and verifying an OT solution to Finnish prosody. And even with the best computational tools, debugging OT constraints is a hard problem.

References

1. Karttunen, L.: The insufficiency of paper-and-pencil linguistics: the case of Finnish prosody. In Butt, M., Dalrymple, M., King, T.H., eds.: Intelligent Linguistic Architectures: Variations on Themes by Ronald M. Kaplan. CSLI Publications, Stanford, California (2006) 287–300
2. Kiparsky, P.: Finnish noun inflection. In Nelson, D., Manninen, S., eds.: Generative Approaches to Finnic and Saami Linguistics: Case, Features and Constraints. CSLI Publications, Stanford, California (2003) 109–161
3. Elenbaas, N.: A Unified Account of Binary and Ternary Stress. Graduate School of Linguistics, Utrecht, Netherlands (1999)
4. Elenbaas, N., Kager, R.: Ternary rhythm and the lapse constraint. Phonology **16** (1999) 273–329
5. Prince, A., Smolensky, P.: Optimality Theory: Constraint Interaction in Generative Grammar. Cognitive Science Center, Rutgers, New Jersey (1993) ROA Version, 8/2002.
6. Kager, R.: Optimality Theory. Cambridge University Press, Cambridge, England (1999)
7. McCarthy, J.J.: The Foundations of Optimality Theory. Cambridge University Press, Cambridge, England (2002)
8. Frank, R., Satta, G.: Optimality theory and the generative complexity of constraint violability. Computational Linguistics **24**(2) (1998) 307–316
9. Karttunen, L.: The proper treatment of optimality in computational phonology. In: FSMNLP'98., Ankara, Turkey, Bilkent University (1998) cmp-lg/9804002.
10. Eisner, J.: Directional constraint evaluation in Optimality Theory. In: Proceedings of the 18th International Conference on Computational Linguistics (COLING 2000), Saarbrücken, Germany (2000) 257–263
11. Kaplan, R.M., Kay, M.: Regular models of phonological rule systems. Computational Linguistics **20**(3) (1994) 331–378
12. Koskenniemi, K.: Two-level morphology. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki (1983)
13. Kaplan, R.M., Newman, P.S.: Lexical resource reconciliation in the Xerox Linguistic Environment. In: ACL/EACL'98 Workshop on Computational Environments for Grammar Development and Linguistic Engineering, Madrid, Spain (1997) 54–61
14. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI Publications, Stanford, CA (2003)