

# Generation of Factorial Typologies in OT: Four Algorithms

Nathan Sanders  
ncsander@indiana.edu

December 1, 2008

## 1 Introduction

This paper explains four algorithms for the generation of factorial typologies in Optimality Theory (OT): naive factorial, bounded factorial, exponential Recursive Constraint Demotion and exponential r-volume. Factorial typologies are an important part of OT. When constructing an OT analysis, linguists use a factorial typology to check the predictions made by the relevant constraints. Candidate sets that are part of the factorial typology should be attested in at least one language of the world. Conversely, candidate sets that are not part of the factorial typology can never win an optimality theoretic evaluation. Therefore, OT predicts that these candidate sets should not appear in any language.

The naive factorial typology algorithm generates all factorial combinations of constraints and then evaluates them using OT's EVAL. This approach is obvious and simple, but is not practical for use with large constraint sets. The naive factorial generation does not use any properties of OT to avoid unnecessary work on the part of EVAL.

The bounded factorial algorithm starts from the naive one; it interleaves factorial generation and evaluation to avoid trying constraint hierarchies that will always produce the same winner set. Interleaving evaluation with generation means that evaluation can model creation of the constraint hierarchy as promotion. Evaluation is as a result incremental, unlike EVAL. The bounded factorial algorithm is inspired by the violation boundaries of Samek-Lodovici and Prince (2002); the bounded factorial call graph is identical to the violation boundary tree. This bounded factorial algorithm is presented for the first time in this paper. Like the naive algorithm, it has

a worst-case factorial time bound, but the average-case time is much better because uninformative constraint set configurations can be skipped.

The exponential Recursive Constraint Demotion algorithm is used in Hayes' OT-Soft (Hayes et al. 2003). It runs Recursive Constraint Demotion (Tesar and Smolensky 1993) on every candidate set—each configuration of the informative output candidates for each input. Recursive Constraint Demotion (RCD) will learn a constraint hierarchy consistent with some candidate set. However, it can be used as a consistency check since it will crash for inconsistent candidate sets for which no constraint hierarchy can be found. These inconsistent candidate sets are not part of the factorial typology. This algorithm avoids the problem of generating all factorial combinations of constraints and has exponential time bounds in the number of candidate sets instead. To my knowledge, this is the first published analysis of this algorithm, although the algorithm is moderately well known and is referenced in other work such as Pater (2008) and used in other software such as Prince's OT Workplace.

The exponential  $r$ -volume algorithm is based on Riggle's (2008)b work on  $r$ -volumes. Riggle's algorithm to calculate  $r$ -volumes is similar to RCD but can terminate faster; it operates incrementally at the individual constraint level in a way similar to the bounded factorial algorithm. This algorithm, like Hayes', evaluates every candidate set, but since its core is the faster  $r$ -volume algorithm, it is correspondingly faster. Riggle first demonstrated the algorithm in a presentation to the Linguistic Society of America (Riggle et al. 2007).

## 1.1 Optimality Theory

Optimality theory is simple, formal model of decision making that consists of strictly ranked violable constraints (Prince and Smolensky 1993). Its restricted computational power has made it attractive in the field of phonology, where its predictions are useful and its simplicity appropriate (Moreton 1996). Although it is used most in the field of phonology, it has been applied to syntax as well. It is related to cognitive models such as neural nets, but is less powerful and builds more structure into the theory.

OT consists of INPUT, a set of words that make up a language  $L$ ; CON, a constraint hierarchy; GEN, a function that takes an input form  $i$  from INPUT and produces a set of output candidates  $C$ ; and EVAL, a function that maps the candidate set  $C$  to a single winner output that, in phonology, is the pronounced form.

The constraint hierarchy CON is a list of constraints that specify the best

/cost#us/		*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a.	cost.us	*!	*			
b.	cos.us		*!			*
c.	☞ cos.tus			*		

Table 1: Tableau for /cost#us/

output candidate from  $C$ . Each constraint is a function of two arguments: the input form and the output candidate. The constraint’s output is the number of violations that the output candidate incurs. EVAL evaluates the constraints for each candidate in  $C$  in the order specified by CON. For each constraint, the optimal candidates, those with the lowest number of violations, are found. All candidates with a higher number of violations are removed. This process continues until all constraints in CON have been evaluated or only one candidate remains. The remaining candidate is the winner.

In standard OT analyses the process is usually captured and presented in a tableau. Each tableau gives the current input, a vertical listing of the output candidates, and a horizontal list of the constraint hierarchy. In the table to the right of the output candidates, each constraint violation for a candidate is marked with a star. The point that EVAL eliminates a candidate is marked with an exclamation point. Since EVAL does not need to consider this candidate further, the background is grey, although violations are still marked. Tableau 1 is an example that will be used throughout the paper.

### 1.1.1 Algorithm

Before giving an implementation of the evaluation algorithm, data structures are needed to model the various parts of OT explained above. These data structures are given in figure 1. The most important type is unsurprisingly **Tableau**, which pairs a list of constraints with a list of rows. Each row pairs a candidate with its violations stored as integers. However, two other representations of tableaux are useful. **CandTableau** stores violations in columns instead of rows. An translation of tableau 1 to **CandTableau** form is given in figure 2. **ConTableau** is a relative tableau (Prince 2002), also column-major. Relative tableaux store relative violations instead of integer violations; relative violations are modeled with the type **Erc**: Elementary Ranking Conditions, which specify only whether constraint favors the desired winner, a loser, or neither.

```

type Tableau = ([Constraint], [Row])
type CandTableau = ([Candidate], [[Int]])
type ConTableau = [Column]

data ERC = L | E | W deriving (Eq, Ord, Show, Read)
type Constraint = String
type Candidate = String
type Column = (Constraint, [ERC])
type Row = (Candidate, [Int])

```

Figure 1: Haskell data types for OT

```

(["cost.us", "cos.us", "cos.tus"],
 [[1,0,0],
  [1,1,0],
  [0,0,1],
  [0,0,0],
  [0,1,0]]) :: CandTableau

```

Figure 2: Haskell value for /cost#us/ tableau

```

eval :: CandTableau -> Candidate
eval (cands, []) = head cands
eval (cands, col:cols) = eval (promote cands (col,cols))

promote :: [Candidate] -> ([Int],[[Int]]) -> CandTableau
promote cands (col,cols) = (filterLosers cands,map filterLosers cols)
  where filterLosers = filterProxy (==minimum col) col

filterProxy p proxy l = map fst (filter (p . snd) (zip l proxy))

```

Figure 3: EVAL implementation

An implementation of the evaluation algorithm is given in figure 3. The input is a column-major `CandTableau`. This representation requires all candidates to be specified and constraint violations assessed; the role of GEN is played by the linguist. An online version of OT with interacting GEN and EVAL is beyond the scope of this paper; see Heiberg’s (1999) dissertation for an implementation.

`eval` takes a column-major `CandTableau`. If all constraints have been evaluated, then all the remaining candidates should be winners. However, this implementation assumes that there is exactly one winner and ignores the rest. If there are still constraints to evaluate, the algorithm removes nonoptimal candidates in `promote` and looks at the next constraint.

`promote` is relatively simple. It finds the minimum violation for the current constraint column and removes all nonoptimal candidates and their associated violation rows. This removal corresponds precisely to those violations marked with an exclamation point in a standard OT tableau, such as tableau 1. Those rows will not be evaluated for succeeding constraints, meaning that the shaded cells of standard tableaux are exactly those eliminated, with the exception of the winner’s row. All constraints are evaluated for the winner. However, adding the line `eval ([cand], cols) = cand` to the beginning of the current code will allow evaluation to finish when only one winner remains, matching the grayed cells exactly.

### 1.1.2 Example

To illustrate factorial typology, Anttila and Andrus (2006) use t-deletion in English. I will use a subset of their exposition to illustrate EVAL in OT. t-deletion in English varies quite a bit between different dialects of the

*COMPLEX	Avoid consonant clusters within a syllable
ONSET	Every syllable must have an onset
ALIGN-LEFT-WORD	Output syllables cannot straddle underlying word boundaries
PARSE	Every underlying segment must appear in the output
ALIGN-RIGHT-PHRASE	Phrase-final input consonants must be syllable-final in the output

Table 2: t-deletion constraints

*COMPLEX(/cost#us/, [cost.us])	*	The final [st] cluster is complex
*COMPLEX(/cost#us/, [cos.us])	√	/t/ deletes, simplifying the word-final cluster to [s]
ONSET(/cost#us/, [cos.us])	*	[us] has no onset.
ONSET(/cost#us/, [cos.tus])	√	/t/ resyllabifies as the onset of [tus]
ALIGN-L-W(/cost#us/, [cos.tus])	*	[cos.t] is not aligned with syllable boundaries
ALIGN-L-W(/cost#us/, [cos.us])	√	t-deletion realigns [cos] with /cost/
PARSE(/cost#us/, [cos.us])	*	Input segment /t/ does not appear in the output
PARSE(/cost#us/, [cos.tus])	√	
ALIGN-R-P(/cost/, [cos])	*	Phrase-final /t/ does not end the syllable
ALIGN-R-P(/cost/, [cost])	√	

Table 3: Constraint evaluation examples

language. For example, /cost#us/ may be produced as [cost.us] or [cos.us]. Anttila and Andrus (2006) analyze the factorial typology of this deletion, matching dialect with deletion pattern and frequency. Their analysis will serve as the basis of a running factorial typology example throughout this paper.

/t/ deletes more often in some phonological contexts than others. Across all dialects, /t/ is most likely to delete before a consonant and least likely before a vowel; word-final deletion is intermediate. There are five constraints necessary to model t-deletion, given in table 2. Examples of satisfaction and violation of the constraints are given in 3. ALIGN-LEFT-WORD and ALIGN-RIGHT-PHRASE are hereafter abbreviated ALIGN-L-P and ALIGN-R-P, respectively.

Given the above input /cost#us/ and the constraints from table 2, tableau (1) of figure 4 is the starting point for EVAL. EVAL first evaluates \*COMPLEX, eliminating [cost.us] and its row since [cost.us] is suboptimal for \*COMPLEX. EVAL moves to ONSET In step (2). This eliminates [cos.us]. Notice that even though [cost.us] also has a violation of ONSET, it not evaluated because it has already been eliminated. If EVAL is enhanced with the early exit clause mentioned above, the algorithm is complete and

1. Evaluate \*COMPLEX

/cost#us/	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. cost.us	*!	*			
b. cos.us		*			*
c. cos.tus			*		

2. Evaluate ONSET

/cost#us/	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
b. cos.us	*!			*
c. cos.tus		*		

3. Evaluate ALIGN-L-W (and ALIGN-R-P and PARSE)

/cost#us/	ALIGN-L-W	ALIGN-R-P	PARSE
c. $\text{[cos.tus]}$ cos.tus	*		

Figure 4: Execution of EVAL

[cos.tus] is the winner. Otherwise, the remaining constraints will evaluate only [cos.tus]. Since it trivially has the minimum constraint violation of its singleton set, it will be the winner for the remaining constraints.

This single-input definition of EVAL extends straightforwardly to sets of input forms. EVAL of multiple inputs is needed to produce a factorial typology, which necessarily involves enumerating multiple winner sets from an set of inputs.

## 1.2 Factorial Typology

The example above finds the winner for a single input and a single constraint ranking. In contrast, a factorial typology takes a set of inputs and returns all sets of winner for all permutations of the constraint set. Factorial typologies are important in linguistic analyses because they add an additional level of certainty to the possibility space that is considered. The typical Optimality Theoretic analysis is different from earlier analyses because it enumerates many possible output candidates for an input form and explains why only the winner surfaces. A factorial typology expands on this analysis by looking at possible constraint hierarchies in which other candidates can win.

1. {[cost.us], [cost.me], [cost]}
2. {[cos.us], [cos.me], [cost]}
3. {[cos.us], [cos.me], [cos]}
4. {[cos.tus], [cost.me], [cost]}
5. {[cos.tus], [cos.me], [cost]}
6. {[cos.tus], [cos.me], [cos]}

Figure 5: T-deletion factorial typology

An incorrect constraint set can overgenerate and produce a factorial typology with winner sets that are unattested in any language. Or it can undergenerate and the factorial typology will lack winner sets that are attested in neighboring languages. In OT, just as changing the posited input form means that the candidates need to be re-evaluated, changing the relevant constraint set means that the factorial typology needs to be re-evaluated.

In the t-deletion example given by Anttila and Andrus (2006), the input set is  $\{/cost/, /cost\#us/, /cost\#me/\}$ . The full factorial typology is given in figure 5. Anttila and Andrus identify dialects of English that match these six variants; for example, the set {[cos.tus], [cos.me], [cost]} appears in Tejano English of San Antonio.

Besides pointing out variations that should be attested for neighboring dialects and languages, a factorial typology exposes two additional structural properties of the data. First, multiple constraint orders can generate the same winner set. These orders can be directly captured by other algorithms covered later in the paper; for a factorial typology the important point is that the typology is much smaller than the full permutation of the constraint set. For example, there are only six winner sets in the t-deletion factorial typology, even though there are  $5! = 120$  permutations of the constraint set.

Second, global phonological patterns can be extracted from a factorial typology. Phonological processes occur in different contexts in the factorial typology; some combinations of deletion never occur. For example, t-deletion in English dialects occurs in the following contexts:

1. Nowhere
2. Before a consonant

3. Before a consonant or a vowel
4. Before a consonant or a word ending
5. Before a consonant, vowel or word ending

Some combinations are impossible; there is no dialect with deletion only before vowels. While some combinations are obvious to a linguist, not all are. The complete factorial typology may contain some surprising entries. The typology does not is a necessary result of the constraints, the input forms and the rules by which OT combines them. There is no creative work required of the linguist.

Building on this, patterns can be found between members of the factorial typology. For example, all dialects that delete /t/ word-finally also delete it before a consonant. All dialects that delete /t/ prevocally also delete it before a consonant. However, there is prevocalic t-deletion will never predict whether or not the dialect also has word-final t-deletion, and vice versa. Anttila and Andrus (2006) give an algorithm to automatically find these implicational universals, calling them t-orders. T-orders, too, are a necessary result of the inputs and structure of OT.

### 1.3 Naive Algorithm

The naive algorithm for finding a factorial typology is fairly simple. It must run EVAL in parallel for the input sets and candidate sets for all permutations of the constraint set. Equation 1 gives a formal definition. Figure 6 gives a more complicated and efficient Haskell implementation. The code should be straightforward except for the functions `nub` . `transpose` which run at the end. `transpose` takes the winners for each input and transposes them into winner sets, after which `nub` eliminates duplicate winner sets.

$$permute(C) = \{x + xs | x \in C, xs \in permute(C - x)\} \quad (1)$$

$$\{EVAL(I, C) | C \in permute(CON)\} \quad (2)$$

## 2 Previous Work

Although the naive algorithm is simple, its running time is always factorial in the size of the constraint set. Faster algorithms are considerably more complicated and draw on much related work in computational OT, such as

```

naive :: [Tableau] -> [[Candidate]]
naive = nub . transpose . map (map eval . factorial . colify)
  where factorial (cands,cols) = map ((,) cands) . permute $ cols
        colify :: Tableau -> CandTableau
        colify (_, rows) = (map fst rows, transpose (map snd rows))

permute :: [t] -> [[t]]
permute [] = [[]]
permute l = [x:ys | (x,xs) <- extractions l, ys <- permute xs]

extractions :: [t] -> [(t, [t])]
extractions l = extract l []
  where extract [] _ = []
        extract (x:xs) prev = (x, prev++xs) : extract xs (x : prev)

```

Figure 6: Naive factorial algorithm

learning algorithms and algorithms that generate all relevant outputs for an input candidate.

## 2.1 Recursive Constraint Demotion

The Recursive Constraint Demotion algorithm (RCD) is not obviously applicable to generation of factorial typologies. RCD is a learning algorithm used to produce a constraint ranking consistent with training evidence in the form of input/output pairs. For each input, one output is labeled the winner, and constraints are recursively demoted until all of the winner’s violations are dominated by worse violations of some loser.

Let us define Recursive Constraint Demotion precisely and give an example. Note: the definition given here allows for multiple constraints per stratum instead of a strictly ranked constraint hierarchy with one constraint per stratum. This means that every multi-constraint stratum stands for all rankings derived from permutations of the stratum. For example, a stratum with three constraints,  $C, D, E$ , actually stands for six possible constraint rankings. The training does not indicate which hierarchy is the correct one, so RCD does not attempt to guess.

### 2.1.1 ERCs

RCD depends heavily on Elementary Ranking Conditions (ERCs). Prince (2002) gives an extensive explanation of ERCs and their use in grounding OT in logic. McCarthy (2002) points out that comparative tableaux, based on ERCs, are useful in OT analyses to ensure that the observed output form is actually favored by the proposed constraint hierarchy and not accidentally bounded by a candidate that actually loses. Comparative tableaux make this property more obvious by representing it explicitly.

An ERC is a vector whose values, unlike rows in an OT tableau, are not integers but values in a three-valued logic: W, L and *e*. Converting a row in an OT tableau to an ERC is straightforward; see equation 3, following the definition from Riggle (2008)b. An ERC makes it very obvious whether the violation profile of an output candidate supports or contradicts evidence for the winning output.

$$erc(a \sim b) = \langle \alpha_1, \dots, \alpha_k \rangle \text{ where } \begin{cases} \alpha_i = & \text{W if } C_i(a) < C_i(b) \\ \alpha_i = & \text{L if } C_i(a) > C_i(b) \\ \alpha_i = & e \text{ if } C_i(a) = C_i(b) \end{cases} \quad (3)$$

Equation 3 defines a function *erc* that takes a winning candidate *a* that defeats candidate *b*.  $C_i$  gives the number of constraint violations for some constraint  $C_i$  in CON. For example, the two-candidate absolute tableau 4 become the single-ERC comparative tableau 5. Since [cos.tus] has fewer violations of ONSET than [cos.us], it is the winning candidate. This is reflected in the ERC as a W. However, [cos.us] has fewer violations on ALIGN-L-W, so the ERC has an L here. Finally, both candidates have the same number of violations (zero) for ALIGN-R-P, so this cell of the ERC is *e*. In figure 7, the Haskell implementation converts an entire tableau to a comparative tableau, given a row that has been specified as the winner.

	/cost#us/	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a.	☞ cos.tus		*		
b.	cos.us	*!			*

Table 4: Two-candidate absolute tableau

Notice that some W must dominate every L in the vector. This ‘some-dominates-every’ pattern is repeated throughout OT and appears in many OT algorithms. ERCs are quite useful because they incorporate context: they compare some losing candidate to the winning output. Since the output

/cost#us/	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
[cos.tus ~ cos.us]	W	L	<i>e</i>	W

Table 5: Single-ERC comparative tableau

```

comparative :: (Row,Tableau) -> (Candidate,ConTableau)
comparative ((cand,viols),(con,tab)) =
  (cand, zip con (transpose . map (sub viols) . snd . unzip $ tab))
  where sub row1 row2 = map classify (zipWith (-) row1 row2)
        classify n | n < 0 = W
                  | n > 0 = L
                  | otherwise = E

```

Figure 7: Create a comparative tableau from an absolute one

is implicit, ERCs created from multiple tableaux in the same language can be combined into a single large comparative tableau. A Haskell implementation of this is given in figure 8.

### 2.1.2 RCD Algorithm

Tesar and Smolensky (1993) and Tesar and Smolensky (1998) give the RCD algorithm in a context of supervised OT learning. In supervised learning, the winning form for some input is given. The learner just has to reorder the constraint hierarchy so that the output form is also the winner. Given the ‘some-dominates-every’ pattern above, the basic learning strategy should be obvious. If the output form is not the winner, constraints that favor the loser must be demoted. In other words, every ERC cell containing L must be moved to the right of a cell containing a W.

RCD chooses to be conservative. Even though each L only needs to be dominated by a single W, it demotes constraints containing Ls past all constraints containing a W. This does not produce a strict domination hi-

```

combine :: [(Candidate,ConTableau)] -> ConTableau
combine extracted =
  zip (map fst cols) (map concat . transpose . map (map snd) $ allcols)
  where allcols@(cols:_) = map snd extracted

```

Figure 8: Merge comparative tableau

/cost#us/	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. × cost.us	*!	*			
b. cos.us		*!			*
c. ↗ cos.tus			*		
/cost#me/					
d. × cost.me	*!				
e. ↗ cos.me					*
f. cos.tme	*!		*		

Table 6: Tableaux for /cost#us/ and /cost#me/

erarchy: it has multi-constraint strata. But the algorithm does not have to choose the best constraint to promote because it promotes all of them.

The second step of RCD is to remove the losing candidates that are eliminated by the most recent demotion. The candidates for which all Ls are now dominated by some W can be removed since they can no longer win. Then, with these candidates removed, the recently promoted constraints are no longer informative—they only serve to dominate the just-eliminated losers—and can be removed as well.

Now the process recurs on the remaining constraints until there are no more constraints to demote. Alternatively, RCD can crash if not enough promotable constraints can be found to dominate the loser-favoring candidates. In this case, the winner set is inconsistent and can not appear in the factorial typology. Since at least one constraint must be promoted at each step, the time bound for RCD is  $O(n^2)$ , or more precisely  $(k^2 - k)/2$  for  $k$  constraints, according to Tesar and Smolensky (1993).

For example, consider the constraint demotion of the combined tableaux for the inputs /cost#us/ and /cost#me/ with the outputs [cost.us] and [cost.me]. The original tableaux are given in table 6. The combined comparative tableau is given in table 7. The current constraint hierarchy incorrectly specifies {[cos.tus], [cos.me]} as the winner set, not {[cost.us], [cost.me]}, so before RCD can run, a comparative tableau must be created with {[cos.tus], [cost.me]} as the desired winner set.

The first step of RCD is to find all constraints with Ls and demote them. For table 7, this is \*COMPLEX and ONSET. Then the remaining constraints, ALIGN-L-W, ALIGN-R-P and PARSE, are promoted to the first stratum and dominated losers removed. For this tableau, ALIGN-R-P dominates no losers, but the other two constraints remove the rest of the candidates because each contributes two Ws. Since the tableau is now empty, the con-

	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. [cost.us ~ cos.us]	L	<i>e</i>	<i>e</i>	<i>e</i>	W
b. [cost.us ~ cos.tus]	L	L	W	<i>e</i>	<i>e</i>
c. [cost.me ~ cos.me]	L	<i>e</i>	<i>e</i>	<i>e</i>	W
d. [cost.me ~ cos.tme]	<i>e</i>	<i>e</i>	W	<i>e</i>	<i>e</i>

Table 7: Comparative tableau for [cost.us] and [cost.me]

	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. [cos.us ~ cost.us]	W	<i>e</i>	<i>e</i>	<i>e</i>	L
b. [cos.us ~ cos.tus]	<i>e</i>	L	W	<i>e</i>	L
c. [cost.me ~ cos.me]	L	<i>e</i>	<i>e</i>	<i>e</i>	W
d. [cost.me ~ cos.tme]	<i>e</i>	<i>e</i>	W	<i>e</i>	<i>e</i>

Table 8: Comparative tableau for [cos.us] and [cost.me]

straints trivially have no Ls, and are all promoted to the next stratum. No constraints remain to demote, so the final ranking is {ALIGN-L-W, ALIGN-R-P, PARSE  $\gg$  \*COMPLEX, ONSET}.

Of course, combining tableaux can produce an inconsistent tableau. The outputs [cost.us] and [cos.me], for example, lead to an inconsistency in which no constraints can be found for promotion because the constraints containing a W all have an L somewhere as well. The comparative tableau is given in table 8. RCD starts promisingly enough; \*COMPLEX, ONSET and PARSE all contain at least one L and must be demoted. That leaves ALIGN-L-W and ALIGN-R-P to be promoted; the Ws in Align-L-W remove candidates (b) and (d), giving the smaller tableau 9.

It is already obvious that we have arrived at a contradiction, but RCD will continue for one more iteration because ONSET has no violations; \*COMPLEX and PARSE will be demoted. However, since ONSET contributes no Ws, no candidates are eliminated. This gives tableau 10.

Again both \*COMPLEX and PARSE are demoted, but there are no constraints left to dominate them, and empty strata are not part of OT, so

	*COMPLEX	ONSET	PARSE
a. [cos.us ~ cost.us]	W	<i>e</i>	L
c. [cost.me ~ cos.me]	L	<i>e</i>	W

Table 9: Comparative tableau for [cos.us] and [cost.me]

	*COMPLEX	PARSE
a. [cos.us ~ cost.us]	W	L
c. [cost.me ~ cos.me]	L	W

Table 10: Comparative tableau for [cos.us] and [cost.me]

```

rcd :: ConTableau -> Bool
rcd cols = case partition (any (==L) . snd) cols of
  (_, []) -> False
  ([], _) -> True
  (demote, promote) -> rcd (demote \*\ promote)
( \*\ ) :: ColTableau -> ColTableau -> ColTableau
demote \*\ promote = alwaysZip titles (transpose filtered)
  where titles = map fst demote
        vlns = transpose (map snd demote)
        filtered = filterProxy (all (==E)) (transpose (map snd
promote)) vlns

alwaysZip first second = zip first (if second==[] then repeat [] else second)
filterProxy p proxy l = map fst (filter (p . snd) (zip l proxy))

```

Figure 9: RCD implementation

RCD crashes with the incomplete hierarchy  $\{\text{ALIGN-L-W}, \text{ALIGN-R-P} \gg \text{ONSET} \gg \}$ . The crash means that the winner set  $\{\text{[cos.us]}, \text{[cost.me]}\}$  is inconsistent with this constraint set and will never appear in any language.

It is this inconsistency check that is useful for generation of factorial typology; inconsistent winner sets cause RCD to crash and can be removed from the factorial typology. The Haskell version of an RCD consistency check is given in figure 9. Note that this algorithm is a consistency check only; it does not return the constraint hierarchy. The two important functions are `rcd` itself, which finds constraints with undominated Ls, and the operator `\*\`, which removes dominated candidates from consideration.

## 2.2 Riggle’s R-Volumes

Riggle’s (2008)b r-volume is quite similar RCD—both operate on a comparative tableau that has had a winner set specified. RCD gives the minimally specific constraint ranking for a winner set if such a ranking exists. The

r-volume is the number of possible constraint rankings that are consistent with the winner set. The result can be obtained from RCD's constraint ranking by finding the product of the factorial of the size of each stratum. Riggle's algorithm finds the r-volume directly. This gives less information than RCD but has a time bound of  $k \log_2 k$  instead of  $(k^2 - k)/2$ .

### 2.2.1 R-Volume Algorithm

Like RCD, the input to the r-volume algorithm is an ERC set of winners, a comparative tableau. After the winner for each tableau is chosen, the ERCs are created by comparing the losers to the winner.

The central idea of the algorithm is to divide each tableau into subtableau and multiply the number of valid rankings from each subtableau to find the total. There are two base cases in this recursive definition. If any violation in a column favors a loser, then that constraint contributes no rankings to the r-volume. If all violations in a column favor the winner, that column contributes  $(k - 1)!$  rankings.

The reasoning for the base cases is not immediately obvious. Each all-W constraint column gives  $(k - 1)!$  possibilities because this constraint can safely be ranked above the other constraints and result in a consistent tableau—it favors the winner on every row. Since there are  $k$  constraints, there are  $(k - 1)!$  ways to rank the other constraints below the all-W one. Similarly, there are zero ways that a constraint containing a loser can be ranked above the other constraints and produce a consistent tableau. With this in mind, the recursive removal step serves to narrow the possible rankings in precisely the way as the promotion step of RCD. The removal step also promotes the candidates and continues ranking the rest of the tableau. However, r-volume's removal step operates on a single constraint, unlike RCD's demotion step. Since it does less work at one time, it can avoid more unnecessary work.

$$r(E^k) = \sum_{0 \leq i \leq k} \begin{cases} 0 & \text{if } \alpha_i = L \text{ for any } \alpha \in E \\ (k - 1)! & \text{if } \alpha_i = W \text{ for every } \alpha \in E \\ r(E - w_i) & \text{otherwise} \end{cases} \quad (4)$$

Equation 4 gives the base case followed by the recursive case, which finds the product of the r-volumes from subtableaux. Here,  $E^k$  is a comparative tableau in which the ERCs are of length  $k$ ; in other words, there are  $k$  constraints.  $E - w_i$  removes ERCs from  $E$  with W in the  $i$ -th coordinate and deletes the  $i$ -th coordinate from the remaining ERCs. This removal step is quite similar to the promotion step in RCD.

To see the r-volume algorithm in action, start with tables 6 and 7, the same data used in the RCD example. Extracting just the ERCs from tableau 7 gives the root of figure 10. The constraint names are no longer important, so they are not shown to save space.

Let us trace the execution of the algorithm. The first two columns of the root add 0 rankings to the r-volume because they contain Ls and must be outranked by some other column. The third column leads to the left-most child. Candidates (b) and (d) are removed because the third constraint favors the winner and promoting it will remove them from consideration. The third constraint is itself removed because it is no longer relevant.

This gives a tableau with the first constraint all L, the second and third constraints all  $e$  and the last constraint all W. The first constraint contributes 0 rankings and the last constraint contributes  $(4 - 1)! = 6$  rankings. The second and third constraint recur but produce identical tableau since neither constraint contains W. This smaller child tableau is similar to its parent—its first constraint is all L, its middle constraint all  $e$  and its last constraint all W. The all-L constraint still contributes 0 rankings, but the all-W constraint now contributes  $(3 - 1)! = 2$  rankings because there are fewer combinations of the remaining constraints. The all- $e$  constraint recurs on the still-smaller tableau at the bottom of figure 10. The results are again identical, but there is now only  $(2 - 1)! = 1$  way to rank the all-W constraint above the rest.

Walking back up the tree, this gives a total of three rankings to each of the all- $e$  constraints in the first child tableau. The total for this tableau is therefore  $3 + 3 + 6 = 12$ . The results of the algorithm are similar for the other two children of the root. It is especially obvious from looking at the second child that much of the computation is shared in this algorithm and that it can be sped up considerably if the function implementing the algorithm is memoized. The difference in this case is the evaluation of 7 tableaux for a memoized function and 12 without memoization.

Like RCD, r-volumes can be used as a consistency check. If the r-volume of a winner set is 0, then there are no rankings for which the winner set can appear, and it is not part of the factorial typology. A function that implements a consistency check version of the algorithm is given in 11. Instead of  $(k - 1)!$ , winners return true, and losers return false.

### 2.3 Harmonic Bounds

The connection of Samek-Lodovici and Prince’s (2002) harmonic bounds to generation of factorial typologies is straightforward. Harmonic bounds are a

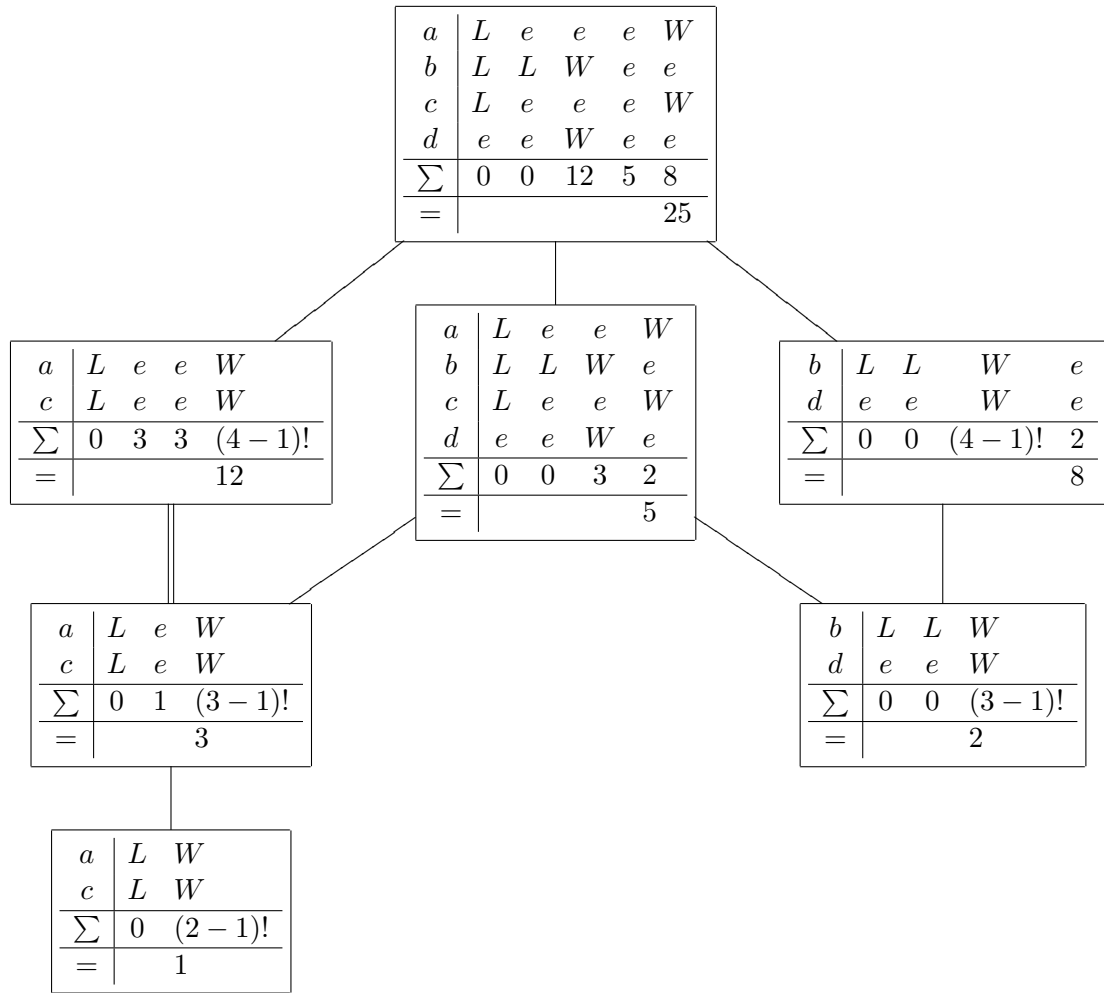


Figure 10: R-Volume for [cost.us] and [cost.me]

```

rVolume :: ConTableau -> Bool
rVolume cols = any r (extractions cols)
  where r (col@(_,erc),cols) | any (==L) erc = False
                             | all (==W) erc = True
                             | otherwise = rVolume (cols \\ col)

(\\) :: [Column] -> Column -> [Column]
cols \\ (title,erc) = alwaysZip titles (transpose filtered)
  where (titles,ercs) = unzip cols
        filtered = filterProxy (/=W) erc (transpose ercs)

alwaysZip first second = zip first (if second==[] then repeat [] else second)
filterProxy p proxy l = map fst (filter (p . snd) (zip l proxy))

```

Figure 11: R-volume implementation

fast way to find the relevant output candidates for some input. Relevant candidates are outputs that can win under some ordering of the constraint set. This is similar to a factorial typology, which is the list of candidate *sets* that can win under some ordering of the constraint set. Harmonic bounds are useful because GEN is theoretically infinite, which is not amenable to computational implementations of OT. However, Samek-Lodovici and Prince (1999) show that the number of relevant candidates for a particular input is bounded by DEP: DEP penalizes unlimited epenthesis, even in cases where some epenthesis is prompted by higher-ranked constraints.

The question is how to sort the relevant candidates from the irrelevant ones that can never win. Knowing that the set is finite does not help one enumerate it. Samek-Lodivici and Prince attack the problem by breaking it into two pieces: a simple harmonic bound (Prince and Smolensky 1993) and a complex bound made from the combination of child bounds. The result is a bounding tree that is worst-case factorial in size.

Simple harmonic bounding is straightforward: a candidate bounds any candidate that has no fewer violations on any constraint and has more violations on some constraint. Here again we see the some-dominates-every structure of OT. For example, in tableau 11, taken from the RCD example in tableau 6, candidate (d), [cost.me] simply bounds candidate (f), [cos.tme]; (f) has a violation of ALIGN-L-W while (d) has none. This is even clearer in the comparative tableau 12: if a candidate's ERC contains no Ls, it is simply bounded by the winner. A simply bounded candidate also contributes no information to any algorithm discussed in this paper.

/cost#me/	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
d. × cost.me	*!				
e. ☞ cos.me					*
f. cos.tme	*!		*		

Table 11: Tableau for /cost#me/

	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
c. [cost.me ~ cos.me]	L	<i>e</i>	<i>e</i>	<i>e</i>	W
d. [cost.me ~ cos.tme]	<i>e</i>	<i>e</i>	W	<i>e</i>	<i>e</i>

Table 12: Comparative tableau for [cost.me]

Complex bounding is nearly as easy to describe, although computation of it is not so simple. To compute it Samek-Lodovici and Prince introduce *bounding sets*. Bounding sets contain candidates that together serve to bound an irrelevant candidate. For each violation in a complex bound, the constraint violation of some member of the bounding set must be less than or equal to the irrelevant candidate. Put another way, a complex bound is a vector which, for each constraint, is one more than the lowest constraint violation of its members, except where all constraint violations are already optimal—that is, identical. In the optimal case, the candidate violation must be less than or equal to the complex bound’s violation. Stated formally, it is

$$\forall i, i \leq n, \forall \alpha \in A, [\lambda(i) < \alpha(i) \rightarrow \exists \alpha', \alpha'(i) < \lambda(i)]. \quad (5)$$

where  $\lambda$  is an irrelevant candidate of length  $n$  and  $A$  is the bounding set. A simple bound becomes a singleton case of a complex bound in which all constraint violations are identical. For example, a bounding set  $\{\langle 011 \rangle\}$  has the bound  $\langle 011 \rangle$ . A complex bounding set  $\{\langle 011 \rangle, \langle 099 \rangle, \langle 010 \rangle\}$  has the bound  $\langle 021 \rangle$ .

Given a number of relevant candidates, classification of a new candidate as relevant or irrelevant is not solved by the introduction of bounding sets. The set of all relevant candidates can create a bound that is too loose. Smaller sets have tighter bounds because of the optimal/singleton bounding case. However, the power set of the relevant candidates is exponential in size and full of redundant bounds. Samek-Lodovici and Prince give an algorithm that recursively finds minimal bounding sets and forms a tree from them.

This tree makes it relatively fast to determine whether a new candidate

is irrelevant. If it is relevant, the tree can be rebuilt to include the new candidate in the relevant set of winners. This leads to a method for finding the full harmonic boundary. Initialize the bounding tree from the fully faithful output candidate. Then new candidates from GEN that are not harmonically bounded by the current bounding tree are added to the relevant set and cause the bounding tree to be rebuilt. The only missing piece is that GEN must provide candidates in an order such that it can tell that further generation will only provide bounded candidates. Work by Heiberg (1999) shows a possible implementation of this, using constraint violations of previous candidates to guide the generation of new ones.

### 2.3.1 Bounding Algorithm

Each node in the bounding tree contains a bounding set and an associated complex bound. The tree starts with a loose, over-large bounding set consisting of all relevant candidates and recursively arrives at singleton bounding sets at the leaves. Since singleton bounding sets simply bound candidates, the leaf bounds are tight. Children are generated from a non-terminal bounding set by examining each constraint in turn and removing candidates that are non-optimal for that constraint. The constraint that is optimized is also removed as it is no longer informative. This is the equivalent to the promotion operation in RCD.

According to Samek-Lodovici and Prince, worst-case size of the bounding tree is exponential in the number of children because the power set of candidates is the worst-case size of the tree. The tree, however, has the potential to include much less of the candidate power set if there is enough information in the candidate violations. Specifically, if the number of candidates is greater than or equal to the number of constraints, the worst case is not possible. This is related to analysis by Riggle (2008)a of the Vapnik-Chervonenkis dimension of OT.

To see this algorithm at work, let us build a bounding tree for tableau 11. This candidate set is interesting because it contains a harmonically bounded candidate. However, this does not prevent Samek-Lodovici and Prince’s algorithm from building a tree, although it may be one level deeper than necessary. The tree that is built is given in figure 12. Only candidate letters, constraint violations and the complex bound are given for each node; constraint names are left out to save space.

The algorithm starts with the full candidate set at the root. The root node’s complex bound is  $\langle 10101 \rangle$ . For each constraint, the complex bound’s violations are the minimum violation of the bounding set’s plus one, except

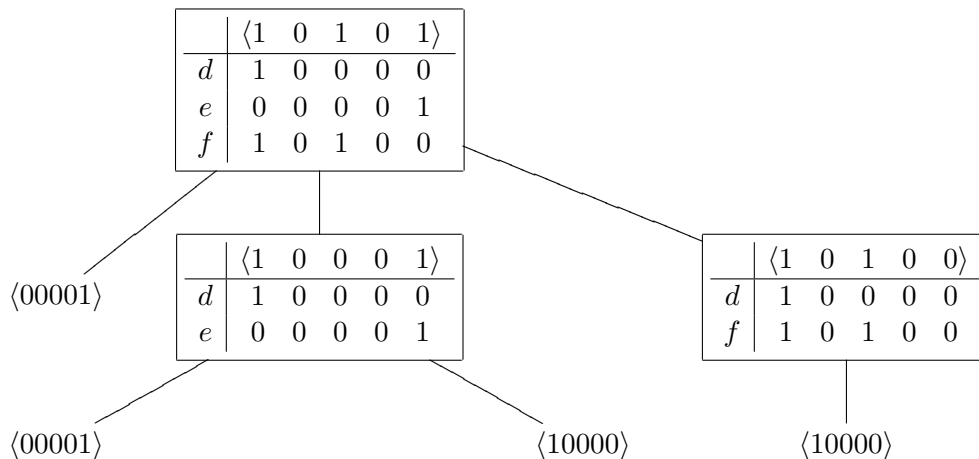


Figure 12: Bounding tree for /cost#me/

constraints for which all members of the bounding set are optimal. In addition, since this is a complex bound, it must recur on each constraint. Only one candidate, [cos.me] (e), is optimal for the first constraint, \*COMPLEX, so the algorithm returns the simple bound  $\langle 00001 \rangle$ , that of candidate (e). The next constraint, ONSET, is the opposite of \*COMPLEX; all candidates are optimal for it. The recursive call is therefore identical to the original call except that ONSET is fixed at 0, so it can be skipped. The third constraint, ALIGN-L-W, is more interesting: it removes candidate (f) but leaves candidates (d) and (e). The recursive call receives a multiple-candidate set and therefore finds another complex bound:  $\langle 10001 \rangle$ .

The rest of the tree is built in much the same way. Notice that the leaves are redundant. Although the bounding tree eliminates many of the redundant bounds, it does not eliminate them all. However, this algorithm only investigated 7 of the possible  $2^5 - 1 = 31$  subsets of the 5 constraints.

### 3 Algorithms

Given the previous work, the factorial typology algorithms are actually quite simple. The implementation of each will be explained in detail. To start, I will present the code for the naive algorithm. It is first given in figure 6 and is repeated in figure 13.

The naive algorithm first generates all permutations of constraint rank-

```

naive :: [Tableau] -> [[Candidate]]
naive = nub . transpose . map (map eval . factorial . colify)
  where factorial (cands,cols) = map ((,) cands) . permute $ cols
        colify :: Tableau -> CandTableau
        colify (_, rows) = (map fst rows, transpose (map snd rows))

permute :: [t] -> [[t]]
permute [] = [[]]
permute l = [x:ys | (x,xs) <- extractions l, ys <- permute xs]

extractions :: [t] -> [(t, [t])]
extractions l = extract l []
  where extract [] _ = []
        extract (x:xs) prev = (x, prev++xs) : extract xs (x : prev)

```

Figure 13: Naive factorial algorithm

ings. Then it evaluates the candidate sets for each ranking—the candidate sets that win make up the factorial typology. The Haskell code implements this in three main steps. The first step is the most involved: evaluation of individual tableaux: `map (map eval . factorial . colify)`. `colify` processes the input tableaux so that it is in column-major form as well as dropping the constraints and splitting the candidates from their violation rows. Then `map eval . factorial` generates all permutations of the constraints and evaluates each one in turn.

Afterward, `transpose` turns the individual winners for each input into winner sets. Finally, `nub` removes duplicates from the typology.

### 3.1 Exponential RCD Algorithm

Hayes’ algorithm is both simple and clever. Instead of evaluating tableaux based on constraints promoted recursively, it takes an entirely different approach. The factorial typology is a series of winner sets, so Hayes generates all possible winner sets and removes inconsistent ones. As described before, RCD can easily provide a consistency check. This method is exponential in the number of input forms and the number of candidates for each input; RCD takes  $(k^2 + k)/2$  time for each winner set but this is dominated by the  $m^n$  time of checking every winner set. The algorithm is well-suited to practical use since most OT data sets have either a small number of input

```

hayes = exponential rcd

exponential :: (ConTableau -> Bool) -> [Tableau] -> [[Candidate]]
exponential test = map (map fst) . filter succeed . sequence . map winners
  where succeed = test . combine
        winners :: Tableau -> [(Candidate,ConTableau)]
        winners (con,rows) =
          [comparative (row,(con,rows)) | (row, rows) <- extractions rows]

comparative :: (Row,Tableau) -> (Candidate,ConTableau)
comparative ((cand,viols),(con,tab)) =
  (cand, zip con (transpose . map (sub viols) . snd . unzip $ tab))
  where sub row1 row2 = map classify (zipWith (-) row1 row2)
        classify n | n < 0 = W
                  | n > 0 = L
                  | otherwise = E

combine :: [(Candidate,ConTableau)] -> ConTableau
combine extracted =
  zip (map fst cols) (map concat . transpose . map (map snd) $ allcols)
  where allcols@(cols:_) = map snd extracted

```

Figure 14: Exponential RCD implementation

forms or a small number of candidates per input. The code to generate all candidate sets is given in figure 14.

There are three important functions in this implementation. The top-level function is `exponential`, which orchestrates the generation and evaluation of the winner sets. `map winners` constructs relative tableaux for each candidate in each tableau. Then the built-in Haskell function `sequence` generates all combinations of these relative tableaux—that is, tableaux for all winner sets. `filter succeed` combines each winner set into a single tableau with `combine` and removes inconsistent winner sets by running RCD as a consistency check. Finally, `map (map fst)` extracts the candidate names from the tableaux.

### 3.2 Exponential R-Volume Algorithm

In figure 14, any consistency check function can be passed to `exponential` as `test`. Since Riggle’s algorithm is identical in relying on `exponential`

it can be written `riggle = exponential rVolume`, given the definition of `rVolume` in section 2.2.1. The only difference is that the performance of Riggle’s algorithm is better than that of Hayes’ because `rVolume` has a time bound of  $O(k \log_2 k)$ , which is better than RCD’s  $O(k^2)$ . However, the overall time bound is still exponential in the number of input candidates:  $O(m^n)$ .

### 3.3 Bounded Factorial Algorithm

The outline of the bounded factorial algorithm is similar to the naive algorithm. However, it avoids unnecessary work by using the bounds of Samek-Lodovici and Prince. This complicates the implementation because the factorial generation and OT evaluation are not separate as in the naive implementation. In fact, harmonic bounds replace `EVAL` to provide partial evaluation. Permutations of the constraint hierarchy are explored incrementally and at each step of the tree’s generation, only informative tableaux are expanded based on candidates remaining after the newly promoted constraints are evaluated.

The two major differences between this algorithm and Samek-Lodovici and Prince’s algorithm to generate a bounding tree is that this algorithm operates over multiple candidates and does not build the tree in memory, although its execution path is a recursive tree. Since there are multiple tableaux, one for each candidate, the bounding algorithm can only quit when all tableaux are uninformative.

To be more concrete, let us look at the code in figure 15. The top-level function `bounded` converts tableaux to column-major form before calling the main recursive function `factStep`. The first clause of `factStep` checks for single-column tableaux. This degenerate base case means that no more information is available from the constraint ranking since the constraint hierarchy is fully ranked—the winning set is available by evaluating the remaining candidates on the remaining constraint in `promoteLast`.

In the normal case, `factStep` investigates each constraint in turn, using the same procedure as for the bounding tree algorithm: only candidates with minimum violation for the constraint in question are used. `step` generates the new, smaller tableaux and skips the ones whose promoted constraints are already optimal and thus uninformative. The tableau sets of `step` are given to the recursive step one at a time. The pieces of factorial typology returned by the recursive step are combined by `Set.unions`.

`recur` is the function that most closely mimics the bounding tree algorithm. Its first clause checks whether all tableaux contain only a single candidate, in which case the recursion is done and the candidate set forms

/cost#us/	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. × cost.us	*!	*			
b. cos.us		*!			*
c. ☞ cos.tus			*		
/cost#me/					
d. × cost.me	*!				
e. ☞ cos.me					*
f. cost.me	*!		*		
/cost/					
g. cost	*!				
h. ☞ cos				*	*

Table 13: Tableaux for /cost#us/, /cost#me/, /cost/

part of the harmonic bound. This means that the candidate set is necessarily part of the factorial typology since members of the harmonic bound must be winner sets to bound irrelevant candidate sets.

If the base case is not satisfied, **recur** eliminates uninformative constraints from the constraint set and recurs on **factStep** again. Uninformative constraints are those for which all candidates are optimal. A constraint whose violations are all 0 or all 1, for example, is uninformative since promoting it does not eliminate any non-optimal candidates.

## 4 Example

The example given by Anttila and Andrus (2006) starts with the inputs /cost#us/, /cost#me/ and /cost/ and the constraint set \*COMPLEX, ONSET, ALIGN-L-W, ALIGN-R-P, and PARSE. The complete tableaux are given in tableau 13.

### 4.1 Exponential RCD

The exponential RCD method first generates all combinations of candidates as winner sets. For the candidates in tableau 13, there are  $3 \cdot 3 \cdot 2 = 18$  winner sets, given in figure 16. The factorial typology for this example is given in figure 17.

For this example, I will continue with two interesting winner sets from figure 16: {[cos.us], [cos.me], [cost]} and {[cost.us], [cost.me], [cos]}. The

```

bounded :: [Tableau] -> [[Candidate]]
bounded = Set.toList . factStep . map colify

factStep :: [CandTableau] -> Set.Set [Candidate]
factStep tabs | any singleCol tabs = Set.singleton (map promoteLast tabs)
factStep tabs = Set.unions . map recur . step $ tabs

recur :: [CandTableau] -> Set.Set [Candidate]
recur tabs | all singleRow tabs = Set.singleton (map (head . fst) tabs)
           | otherwise = factStep (withCands informative tabs)

step :: [CandTableau] -> [[CandTableau]]
step tabs = map (map (uncurry promote) . zip candsets) tabsets
  where candsets = map fst tabs
        tabsets = check . transpose . map (extractions . snd) $ tabs
        check = filter (any (notSame . fst))

informative :: [[[Int]]] -> [[[Int]]]
informative violations = map (filterProxy id informativeCols) violations
  where informativeCols = map (any notSame) (transpose violations)

promoteLast :: CandTableau -> Candidate
promoteLast (cands,col:cols) = head . fst $ promote cands (col, [])

notSame = any (uncurry (/=)) . win2
singleRow = (==1) . length . head . snd
singleCol = (==1) . length . snd
withCands f tabs = zip (map fst tabs) (f (map snd tabs))

win2 l = zip l (tail l)

```

Figure 15: Bounded factorial implementation

1. {[cost.us],[cost.me],[cost]}
2. {[cost.us],[cost.me],[cos]}
3. {[cost.us],[cos.me],[cost]}
4. {[cost.us],[cos.me],[cos]}
5. {[cost.us],[cos.tme],[cost]}
6. {[cost.us],[cos.tme],[cos]}
7. {[cos.us],[cost.me],[cost]}
8. {[cos.us],[cost.me],[cos]}
9. {[cos.us],[cos.me],[cost]}
10. {[cos.us],[cos.me],[cos]}
11. {[cos.us],[cos.tme],[cost]}
12. {[cos.us],[cos.tme],[cos]}
13. {[cos.tus],[cost.me],[cost]}
14. {[cos.tus],[cost.me],[cos]}
15. {[cos.tus],[cos.me],[cost]}
16. {[cos.tus],[cos.me],[cos]}
17. {[cos.tus],[cos.tme],[cost]}
18. {[cos.tus],[cos.tme],[cos]}

Figure 16: Winner sets for /cost#us/, /cost#me/, /cost/

1. {[cost.us], [cost.me], [cost]}
2. {[cos.us], [cos.me], [cost]}
3. {[cos.us], [cos.me], [cos]}
4. {[cos.tus], [cost.me], [cost]}
5. {[cos.tus], [cos.me], [cost]}
6. {[cos.tus], [cos.me], [cos]}

Figure 17: Factorial typology for /cost#us/, /cost#me/, /cost/

	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. [cos.us ~ cost.us]	W	<i>e</i>	<i>e</i>	<i>e</i>	L
b. [cos.us ~ cos.tus]	<i>e</i>	L	W	<i>e</i>	L
c. [cos.me ~ cost.me]	W	<i>e</i>	<i>e</i>	<i>e</i>	L
d. [cos.me ~ cos.tme]	W	<i>e</i>	W	<i>e</i>	<i>e</i>
e. [cost ~ cos]	L	<i>e</i>	<i>e</i>	W	W

Table 14: Comparative tableau for [cos.us], [cos.me], [cost]

first set is part of the factorial typology, even though it is not obvious that t-deletion can happen prevocally but not word-finally. The second set is not part of the factorial typology, even though it is not obvious that word-final t-deletion requires interconsonantal t-deletion.

The relative tableau for the first winner set example is tableau 14. Once the relative tableau is created, RCD must check that the winner set is consistent. A consistent winner set allows RCD to promote all constraints. The first step is shown in tableau 15. Here, ALIGN-L-W and ALIGN-R-P have been promoted because they contain no L cells. The candidates with W cells for ALIGN-L-W and ALIGN-R-P have also been removed since they are now bounded by these two promoted constraints.

	*COMPLEX	ONSET	PARSE
a. [cos.us ~ cost.us]	W	<i>e</i>	L
c. [cos.me ~ cost.me]	W	<i>e</i>	L

Table 15: RCD of [cos.us], [cos.me], [cost], step 2

	*COMPLEX	ONSET	ALIGN-L-W	ALIGN-R-P	PARSE
a. [cost.us ~ cos.us]	L	<i>e</i>	<i>e</i>	<i>e</i>	W
b. [cost.us ~ cos.tus]	L	L	W	<i>e</i>	<i>e</i>
c. [cost.me ~ cos.me]	L	<i>e</i>	<i>e</i>	<i>e</i>	W
d. [cost.me ~ cos.tme]	<i>e</i>	<i>e</i>	W	<i>e</i>	<i>e</i>
e. [cos ~ cost]	W	<i>e</i>	<i>e</i>	L	L

Table 16: Comparative tableau for [cost.us], [cost.me], [cos]

	*COMPLEX	ONSET	ALIGN-R-P	PARSE
a. [cost.us ~ cos.us]	L	<i>e</i>	<i>e</i>	W
c. [cost.me ~ cos.me]	L	<i>e</i>	<i>e</i>	W
e. [cos ~ cost]	W	<i>e</i>	L	L

Table 17: RCD of [cost.us], [cost.me], [cos], step 2

The process ends on the next step. First \*COMPLEX and ONSET are promoted since they contain no L cells. Now \*COMPLEX bounds candidates (a) and (c), and they are removed. There are now no candidates left in the tableau and PARSE is trivially promotable; it contains no L cells because it contains no cells at all. Therefore, {[cos.us], [cos.me], [cost]} is part of the factorial typology.

Let us now look at {[cost.us], [cost.me], [cos]}. Its initial comparative tableau is 16. The first step of RCD promotes only ALIGN-L-W and removes candidates (b) and (d). This gives tableau 17. Now ONSET is promoted, but this removes no candidates since it has no W cells. None of the remaining constraints, (\*COMPLEX, ALIGN-R-P and PARSE) can be promoted since they all contain at least one L. RCD fails and as a result the winner set {[cost.us], [cost.me], [cos]} is not part of the factorial typology.

## 4.2 Exponential R-volume

Riggle’s method is identical to Hayes’ except for the use of r-volume as a consistency check instead of RCD. Therefore, this example will start from the same two winning sets as those in the Hayes example. The r-volume algorithm starts with tableau 14. It skips \*COMPLEX and ONSET because they contain at least one L. Next, since ALIGN-L-W contains some W cells but not all Ws, the algorithm removes ALIGN-L-W and candidates (b) and (d) to produce tableau 18.

Here, \*COMPLEX is again skipped because it still contains an L for can-

	*COMPLEX	ONSET	ALIGN-R-P	PARSE
a. [cos.us ~ cost.us]	W	<i>e</i>	<i>e</i>	L
c. [cos.me ~ cost.me]	W	<i>e</i>	<i>e</i>	L
e. [cost ~ cos]	L	<i>e</i>	W	W

Table 18: R-volume consistency check for {[cos.us], [cos.me], [cost]}, call 2

	*COMPLEX	ALIGN-R-P	PARSE
a. [cos.us ~ cost.us]	W	<i>e</i>	L
c. [cos.me ~ cost.me]	W	<i>e</i>	L
e. [cost ~ cos]	L	W	W

Table 19: R-volume consistency check for {[cos.us], [cos.me], [cost]}, call 3

didate (e), but ONSET now contains only *e* cells, which causes another recursive call. However, no candidates are removed because ONSET has no W cells. This produces tableau 19. Now \*COMPLEX is skipped yet again, and ALIGN-R-P causes a recursive call after removing candidate (e). In tableau 20, we finally see that \*COMPLEX consists of nothing but W cells. This means that the r-volume is positive, the candidate set is consistent, and that it is part of the factorial typology.

For inconsistent candidate sets, the r-volume consistency check may be forced to do more work since it can not quit early upon finding a positive r-volume. However, in the case of the candidate set {[cost.us], [cost.me], [cos]}, the recursion is quite restricted because most constraints contain an L and therefore can be discarded immediately. The starting tableau 16 is the same as that of RCD for the same candidate set.

The first iteration of r-volume eliminates all constraints but ALIGN-L-W, since it is the only one that does not contain an L. Recursion on ALIGN-L-W creates tableau 21 with candidates (b) and (d) removed. Here the situation is almost identical: all constraints except ONSET contain an L, so recursion happens only on ONSET, giving tableau 22.

Since ONSET has no Ws, it removes no candidates and as a result every

	*COMPLEX	PARSE
a. [cos.us ~ cost.us]	W	L
c. [cos.me ~ cost.me]	W	L

Table 20: R-volume consistency check for {[cos.us], [cos.me], [cost]}, call 4

	*COMPLEX	ONSET	ALIGN-R-P	PARSE
a. [cost.us ~ cos.us]	L	<i>e</i>	<i>e</i>	W
c. [cost.me ~ cos.me]	L	<i>e</i>	<i>e</i>	W
e. [cos ~ cost]	W	<i>e</i>	L	L

Table 21: R-volume consistency check for {[cost.us], [cost.me], [cos]}, call 2

	*COMPLEX	ALIGN-R-P	PARSE
a. [cost.us ~ cos.us]	L	<i>e</i>	W
c. [cost.me ~ cos.me]	L	<i>e</i>	W
e. [cos ~ cost]	W	L	L

Table 22: R-volume consistency check for {[cost.us], [cost.me], [cos]}, call 3

constraint in tableau 22 is discarded. The r-volume algorithm fails to find an all-W constraint anywhere in its search tree, meaning that the r-volume is zero. This means that {[cost.us], [cost.me], [cos]} is inconsistent and not part of the factorial typology.

### 4.3 Bounded Factorial

The bounded factorial algorithm requires more exposition than the other two because its execution cannot be easily subdivided in the same way as the candidate set enumeration approach. Instead, portions of the entire search tree are given below: the first, figure 18, shows the initial call and its children. The second and third figures, 19 and 20, show the complete search tree for ALIGN-L-W and PARSE. These constraints produce search trees intermediate in size; that of \*COMPLEX is smaller and those of ALIGN-R-P and ONSET are larger.

The notation is somewhat abbreviated to fit the call tree into available space. Candidates are identified by their letters and constraints are identified by a single capital letter, the first except for ALIGN-L-W and ALIGN-R-P, which would otherwise be ambiguous. Constraint violations are marked in the usual way with an asterisk. Each tableau represents a call in the call tree; the children represent recursive calls. The number in the top-left of each tableau is the indices of the constraints that have been promoted.

Figure 18 shows algorithm's first step, starting from the root node. Each constraint in turn is removed as well as any candidates that are not optimal for that constraint. This is equivalent to promoting the constraint above the remaining constraints and below any previously-promoted constraints and

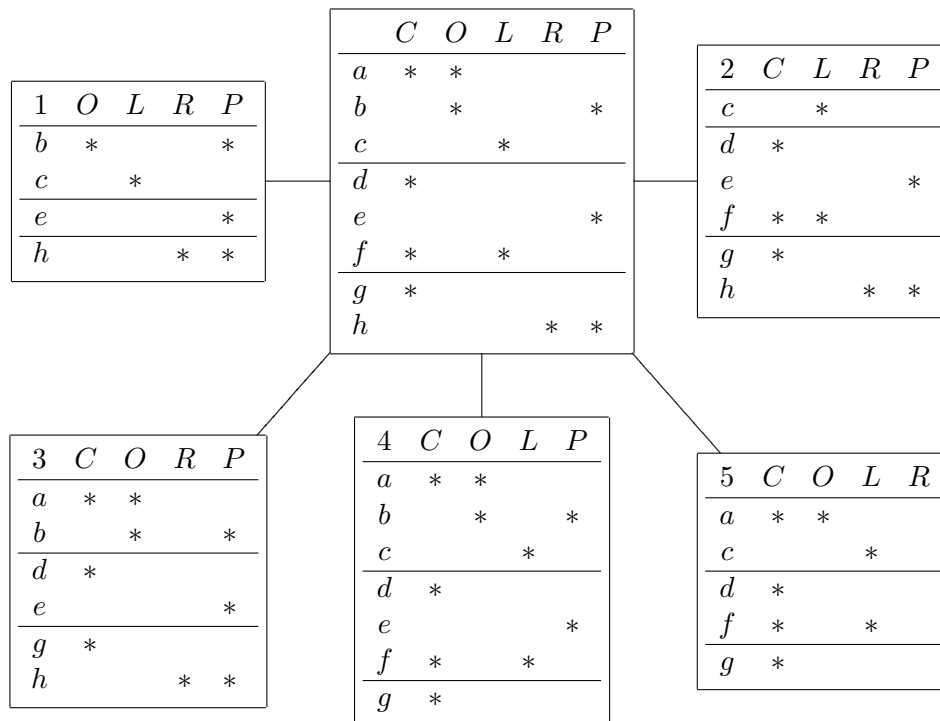


Figure 18: Bounded factorial call tree for [cos.us], [cos.me], [cost]

in fact calls `promote` in the implementation. For the root node, 5 children result. For example, when `*COMPLEX` is promoted (producing tableau 1 in the figure), candidates (a), (d), (f) and (g) are removed because they are not optimal with respect to `*COMPLEX`. At this point, the outputs for `/cost#me/` and `/cost/` have winners: (e) and (h). However, for `/cost#us/` (c) will only win if `ALIGN-L-W` is promoted. If `ONSET` or `PARSE` are promoted, (b) will win. `ALIGN-R-W` is already optimal for all candidates and is skipped.

Figure 19 picks up evaluation from child tableau 3 of figure 18, which promotes `ALIGN-L-W`. Its first child tableau, tableau 31, promotes `*COMPLEX`, which is a terminal tableau because each subtableau has only one candidate. Its winner set is  $\{(b),(e),(h)\}$  or  $\{[cos.us], [cos.me], [cos]\}$ .

The second child tableau should be 32, the promotion of `ONSET`. However, `ONSET` is skipped because it is not informative: all candidates are equally optimal with respect to `ONSET`. Instead, the second child tableau is

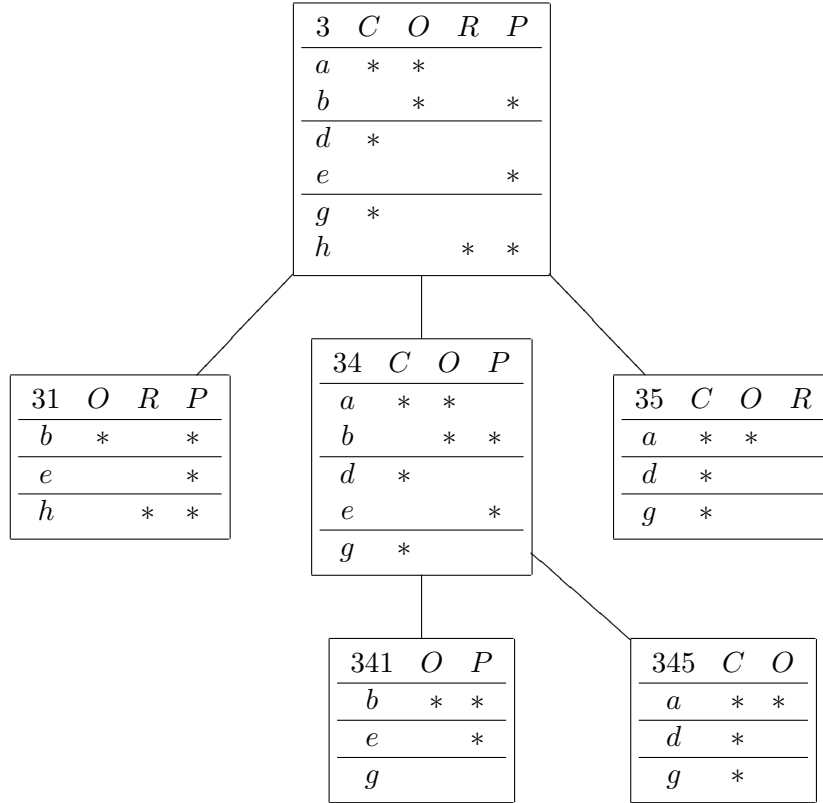


Figure 19: Bounded factorial call tree, ALIGN-L-W branch

34, the promotion of ALIGN-R-P, which eventually leads to two winner sets,  $\{[\text{cos.us}], [\text{cos.me}], [\text{cost}]\}$  and  $\{[\text{cost.us}], [\text{cost.me}], [\text{cost}]\}$ . Finally, promoting PARSE gives tableau 35 and the winner set  $\{[\text{cos.us}], [\text{cos.me}], [\text{cost}]\}$  again.

Notice here that promoting ALIGN-R-P does redundant work in the subsequent promotion of PARSE in tableau 345. Unfortunately this algorithm cannot avoid all redundant work so this winner set is calculated twice. Redundant work also happens when two promotions lead to the same tableau, such as in tableaux 53 and 35. Memoization of the algorithm on the candidate and constraint sets could avoid this redundancy.

Figure 20 shows the call tree that results from promoting PARSE. It is similar to figure 19. One difference is that at the bottom of the search tree tableau 5123 triggers the single-constraint terminal clause of `factStep`.

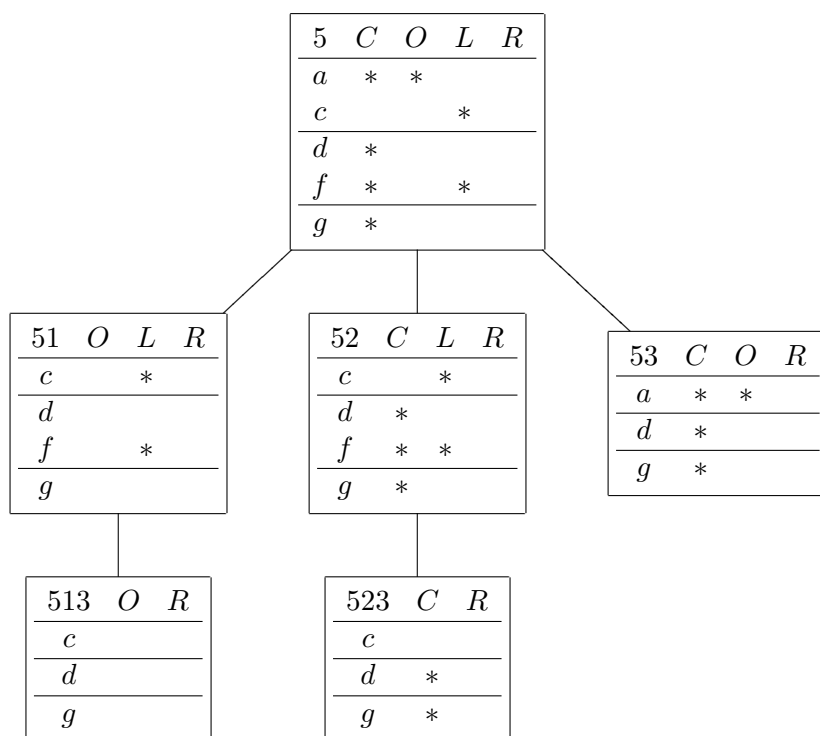


Figure 20: Bounded factorial call tree, PARSE branch

There is only one winner for each subtableau in 5123. In the case that some subtableau has more than one winner even for a single-constraint tableau, the algorithm chooses the first winner that is optimal for the remaining constraint. This case is handled by the function `promoteLast` in the implementation.

## 5 Discussion

Structurally, all three algorithms are quite similar. They all operate by selectively promoting constraints and eliminating candidates. This gives them a common ‘some-W-dominates-every-L’ pattern that repeats throughout. This commonality results from the basic structure of Optimality Theory.

Although the internals of the algorithms are similar, the high level approach is quite different between Hayes and Riggle’s winner-set algorithm and the bounded factorial algorithm given here. Generating all winner sets is a clever and pragmatic attack on the problem; most real world data sets have a small number of inputs and candidates, since linguists generally select only interesting, non-redundant candidates for inclusion. Even better, the method reuses Recursive Constraint Demotion, a well understood algorithm, with only a few changes. And, as Riggle has shown, the method can use any consistency test.

The approach of the bounded factorial algorithm is more straightforward: generate a factorial typology armed with knowledge of OT. Modeling the factorial re-ordering of constraints as promotion allows the algorithm to eliminate irrelevant candidates until each tableau is itself minimal. Unfortunately, this results in a more complicated algorithm.

Of the two approaches, the bounded factorial algorithm is, on the surface, theoretically superior. Its worst-case run time is exponential in the number of constraints, but evidence provided a sufficient number of candidates makes the actual number of calls smaller. On the other hand, Hayes’ winner-set generation is exponential in the number of inputs; there is no obvious theoretical limit on the number of inputs. However, Riggle (2008) recently gave a limit based on the Vapnik-Chervonenkis dimension (VCD), showing that the VCD of OT ensures that the number of non-redundant input forms is linear in the size of the constraint set. Briefly, by using the VCD of OT, he shows that an optimal learner will need at most  $k - 1$  inputs to learn the order of  $k$  constraints. This means that both approaches are exponential in the number of constraints.

In the example given above, performance is similar as well: the bounded

factorial function evaluates 56 tableaux for the example provided above. This is less than half of  $5! = 120$ , but the tableaux at later stages are much smaller than the original and the algorithm does less work than EVAL would for each tableau. On the other hand, Hayes' method with RCD only evaluates 43 progressively smaller tableaux, although each tableaux requires more work. Finally, Riggle's method with r-volume evaluates 96 tableaux, but does less work for each tableau than either bounded factorial or RCD.

All three algorithms are faster than the naive one. Not only do they use properties of OT to reduce the number of tableaux evaluated, they also substitute partial evaluation instead of full EVAL: partial evaluation allows them to remove irrelevant candidates after each promotion. These smaller tableaux are passed on to children in the search tree in the case of the r-volume and bounded factorial algorithms.

## References

- Anttila, Arto, and Curtis Andrus. 2006. T-orders. Technical Report 873, Stanford University.
- Hayes, Bruce, Bruce Tesar, and Kie Zuraw. 2003. Otsoft 2.1. software package, <http://www.linguistics.ucla.edu/people/hayes/otsoft/>.
- Heiberg, Andrea. 1999. *Features In Optimality Theory: A Computational Model*. PhD thesis, University of Arizona.
- McCarthy, John. 2002. *A Thematic Guide to Optimality Theory*. Cambridge University Press.
- Moreton, Elliot. 1996. Non-computable functions in Optimality Theory. Unpublished ms., Available on the Rutgers Optimality Archive as ROA-364.
- Pater, Joe. 2008. Optimization and linguistic typology. Unpublished ms., Ms, University of Massachusetts, Amherst.
- Prince, Alan. 2002. Entailed ranking arguments. Technical Report ROA-500, Rutgers Optimality Archive.
- Prince, Alan, and Paul Smolensky. 1993. Optimality theory: Constraint interaction in generative grammar. Technical Report RuCCS-TR-2, Rutgers University Center for Cognitive Science, New Brunswick, NJ.

- Riggle, Jason. 2008a. The complexity of ranking hypotheses in optimality theory. *Computational Linguistics* 34(4). Forthcoming.
- Riggle, Jason. 2008b. Counting rankings. In B. Joseph (Ed.), *Proceedings of the Linguistic Society of America*, Chicago, January.
- Riggle, Jason, Maximilian Bane, James Kirby, and Jeremy O'Brien. 2007. Efficiently computing OT typologies, January. Linguistic Society of America 2007 Annual Conference in Anaheim, CA.
- Samek-Lodovici, Vieri, and Alan Prince. 1999. Optima. Technical Report RuCCS-TR-57, Rutgers Center for Cognitive Science. Available on the Rutgers Optimality Archive as ROA-363.
- Samek-Lodovici, Vieri, and Alan Prince. 2002. Fundamental properties of harmonic bounding. Technical Report RuCSS-TR-71, Rutgers Center for Cognitive Science. Available on the Rutgers Optimality Archive as ROA-785.
- Tesar, Bruce, and Paul Smolensky. 1993. Learnability in optimality theory: an algorithm and some basic complexity results. Unpublished ms., Available as ROA number 2.
- Tesar, Bruce, and Paul Smolensky. 1998. Learnability in optimality theory. *Linguistic Inquiry* 29:229–268.