Evaluating Precedence-Based Phonology: Logical structure of reduplication and linearization

Hossep Dolatian Eric Raimy

December 7, 2020

Abstract

Historically, reduplication is a challenging phenomenon which has been modeled with diverse mathematical and theoretical frameworks. In this chapter, we focus on total reduplication. We examine the computational expressivity of total reduplication in terms of Monadic-Second Order logic. We then contrast this with an explicit theory of reduplication, Precedence-Based Phonology (Raimy 2000a). We show that this theory of reduplication is not MSO-definable, making it computationally more complex or expressive than reduplication itself. We extrapolate the reasons for this divergence as due to counting.

Contents

1 Introduction

Reduplication is a common morphological process with a wide-ranging typology. In generative phonology and morphology, there have been many competing theoretical formalizations of reduplication, couched in multiple frameworks (Wilbur 1973; Marantz 1982; McCarthy and Prince 1995; Raimy 2000a; Struijke 2000; Inkelas and Zoll 2005; Urbanczyk 2011; Carrier 1979). But in contrast to the plethora of theories for reduplication, there is a dearth of mathematical and computational work on reduplication. This is because reduplication is computationally different from the rest of morphology and phonology. In this paper, we examine the generative capacity of reduplication. We show that it is definable with Monadic-Second Order logic. With this result, we then evaluate a theory of reduplication and show that it is expressive and are more powerful than reduplication.

The organization of this paper follows our argument. In §2, we go over the computational and mathematical properties of reduplication in terms of generative capacity. In computational morphology, it is common knowledge that total reduplication is not a finite-state language (Culy 1985). It cannot be recognized by finite-state acceptors nor generated by a 1-way finite-state transducers (Roark and Sproat 2007; Chandlee 2017). Instead, the language of total reduplication requires a Multi-Context-Free grammar (MCFG) or arguably a Parallel MCFG (PMCFG (Seki et al. 1991, 1993; Albro 2005; Stabler 2004; Clark and Yoshinaka 2014). And as a function, total reduplication is a regular function that is definable with 2-way FSTs (Dolatian and Heinz 2020), which themselves are equivalent to string-to-string functions that use Monadic Second Order logic (MSO). We define a logical transduction for reduplication and triplication.

Having defined an explicit logical transduction for reduplication as a phenomenon, we determine if theories of reduplication are computationally more complex than reduplication. In §3, we note the limited work on formalizing different theories of reduplication. As an in-depth case study in §4, we first focus on Precedence-Based Phonology (PBP), a theoretical framework which exploits graph-based representations to model reduplication (Raimy 1999, 2000a,b). To evaluate its generative capacity in §5, we translate PBP into an explicit computational framework with logical transductions (Courcelle 1997; Engelfriet and Hoogeboom 2001). We show that that PBP is not MSO-definable, making PBP more expressive than reduplication itself.

We briefly discuss this result in the conclusion §6. We generalize this negative result and hypothesize that it is due the counting aspect in PBP. The reliance on counting is clearer for cases like triplication. Future work will look at a larger array of reduplicative theories in order to determine their generative capacity. In the appendix §A, we provide a formal proof that PBP is not MSO-definable.

2 Background in computational morphology and reduplication

A major strand of work in computational linguistics is concerned with finding the exact class of grammars which can model all attested linguistic processes. In other words, This goal is thus about finding the generative capacity of a grammar and contrasting with the generative capacity of its linguistic objects. In this section, we go over the generative capacity of morphology and contrast it with reduplication.

2.1 Most morphology and phonology are regular

It is well-established that most phonological and morphological patterns can be modeled with finite-state calculus (Johnson 1972; Koskenniemi 1983; Gazdar and Pullum 1985; Ritchie 1989, 1992; Kaplan and Kay 1994; Beesley and Karttunen 2003; Roark and Sproat 2007). They thus have a low generative capacity. In terms of well-formedness conditions, most phonological and morphological patterns can be modeled with simple finite-state acceptors (FSAs). Similarly in terms of transformations, phonological and morphological processes can be modeled with finite-state transducers (FSTs). Both of these finite-state machines (FSM) are technically 1-way FSMs which traverse the input in only one direction. In terms of their string languages and functions, morpho-phonological patterns form regular languages while morpho-phonological processes form rational functions.[1] Furthermore, most morpho-phonological patterns and processes can be computed by much weaker computational resources, i.e., as *subregular* classes (Chandlee 2014; Rogers and Pullum 2011; Heinz and Idsardi 2013; Rogers et al. 2013).

Because phonology and morphology are finite-state definable, it is not surprising that most theories of phonology and morphology are also finite-state definable (Karttunen 2003; Roark and Sproat 2007).[2] In fact, there is a recent strand of work in computational phonology which shows that multiple phonological theories are inter-translatable using logical transductions. These theoretical equivalences are found in general phonotactics (Graf 2010a,b), syllabification (Strother-Garcia 2019), and tone (Danis and Jardine 2019; Jardine et al. 2020; Oakden 2020). This chapter is close to this line of work. As we discuss below, we show that most reduplication theories have unclear generative capacity, computational complexity, or expressivity. But at least one such theory is definitely more expressive than reduplication.

2.2 Reduplication is more powerful

In contrast to most of morphology and phonology, certain types of reduplication cannot be modeled with 1-way FSAs or 1-way FSTs. This is because 1-way FSMs cannot memorize an unbounded amount of information about the input string. There are two types of reduplication: partial and total. Partial reduplication involves copying a substring of bounded size (1a). Partially reduplicated strings can be computed by 1-way FSAs, while the process of partial reduplication can be computed by 1-way FSTs (Chandlee and Heinz 2012). However, total reduplication (1b) cannot be modeled with 1-way FSAs or 1-way FSTs because there is no bound on the size of the reduplicant (Culy 1985).

(1)	a.	takki→tak∼takki	‘leg’→‘legs’	*Agta* (Moravcsik 1978:311)
	b.	wanita→wanita∼wanita	‘woman’→‘women’	*Indonesian* (Cohn 1989:308)

The output language of reduplication is not a regular language. Thus, more expressive grammars are needed to recognize or generate totally reduplicated strings. To do so, various augmentations of Context-Free Grammars (CFGs) have been proposed (Manaster-Ramer 1986; Savitch 1989). The most well-developed

[1]Rational functions are a subclass of regular functions. The former are computed by 1-way FSTs while latter by 2-way FSTs (Filiot and Reynier 2016). Kaplan and Kay (1994) depart from convention by calling the functions generated by 1-way FSTs as ‘regular functions.’

[2]The exception is Optimality Theory, which has many finite-state approximations (Eisner 1997, 2000a,b; Karttunen 1998; Frank and Satta 1998; Riggle 2004; Gerdemann and Van Noord 2000; **?**), though the general consensus is that it is not finite-state definable (Idsardi 2006; Heinz et al. 2009; Payne et al. 2017; Hao 2019).

augmentation is a Multiple Context-Free Grammar (MCFG, Seki et al. 1991, 1993) which has been applied for reduplication (Albro 2000, 2005). There is recent work on using the more expressive class of Parallel MCFGs in order to model morphological reduplication and syntactic copying (Kobele 2006; Clark and Yoshinaka 2012, 2014; Clark 2017).

As for applying total reduplication to a string, Dolatian and Heinz (2018) show that total reduplication requires the additional power of 2-way FSTs.[3] These are FSTs which can traverse the input string in multiple directions (Hopcroft and Ullman 1969; Savitch 1982; Engelfriet and Hoogeboom 2001; Filiot and Reynier 2016). In terms of string functions, these 2-way FSTs compute regular functions. Thus, computationally, reduplication is different from the rest of morpho-phonology. Whereas partial reduplication and **all** other morpho-phonological processes can be computed by 1-way FSTs, total reduplication requires more expressive 2-way FSTs.[4]

2.3 Reduplication is MSO-definable

In terms of formal logic, 1-way FSTs compute string-to-string functions that can be described with order-preserving Monadic Second Order logic (MSO), while 2-way FSTs compute string-to-string functions that require non-order-preserving MSO logic (Büchi 1960; Engelfriet and Hoogeboom 2001; Filiot and Reynier 2016).[5] To illustrate, consider the hypothetical input-output pair *kæt∼kæt∼kæt.* We use superscripts on the nodes to mark what copy they belong to.

Figure 1: Total reduplication of *kæt∼kæt*

(a) Input

$$\begin{tikzcd}
\text{k} \arrow[r, "\lhd"] & \text{æ} \arrow[r, "\lhd"] & \text{t}
\end{tikzcd}$$

(b) Output

$$\begin{tikzcd}
\text{k}_1 \arrow[r, "\lhd"] & \text{æ}_1 \arrow[r, "\lhd"] & \text{t}_1 \arrow[dll, "\lhd"] \\
\text{k}_2 \arrow[r, "\lhd"] & \text{æ}_2 \arrow[r, "\lhd"] & \text{t}_2
\end{tikzcd}$$

To generate the right output, our logical transduction uses a copy set of size 2, where each Copy is dedicated to an instance (copy) of *kæt.* The predicates in (2a) find the first and last segment in the input. We need two simple output functions to generate segments in the first copy and second copy. Three output functions connect all the segments via the binary relation of immediate successor ($\lhd$).[6] For a primer on the use of logical functions, see Strother-Garcia (2018). For more information, see Strother-Garcia (2019) or Dolatian (2020).

[3]There are many 1-way FST approximations of total reduplication (Walther 2000; Beesley and Karttunen 2000, 2003; Cohen-Sygal and Wintner 2006; Hulden and Bischoff 2009; Hulden 2009). However, these approximations are generally inadequate (Roark and Sproat 2007:54). See Dolatian and Heinz (2020) for an overview.

[4]For modeling partial reduplication, Dolatian and Heinz (2018) further argue that 2-way FSTs are better suited than 1-way FSTs. This is an argument based on strong generative capacity (Miller 1999; Dolatian and Heinz 2020). We set aside this nuance.

[5]We do not explain the property of order-preservation because it is not crucial for this chapter. This property is salient in other areas of computational morphology (Dolatian 2020) and phonology (Chandlee and Jardine 2019). See Dolatian (2020:ch7) for explanation.

[6]For a given node or variable, the graphs mark copy membership via subscript while the logical formulas use superscripts.

(2) *MSO transduction to apply total reduplication*

a.
- $\mathbf{first}(x) \overset{\text{def}}{=} \neg\exists y[\mathsf{succ}(y,x)]$
- $\mathbf{last}(x) \overset{\text{def}}{=} \neg\exists y[\mathsf{succ}(x,y)]$

b. For every $\mathsf{lab} \in L$
- $\phi\mathtt{lab}(x^1) \overset{\text{def}}{=} \mathsf{lab}(x)$
- $\phi\mathtt{lab}(x^2) \overset{\text{def}}{=} \mathsf{lab}(x)$

c.
- $\phi\mathtt{succ}(x^1, y^1) \overset{\text{def}}{=} \mathsf{succ}(x,y)$
- $\phi\mathtt{succ}(x^1, y^2) \overset{\text{def}}{=} \mathbf{last}(x) \wedge \mathbf{first}(y)$
- $\phi\mathtt{succ}(x^2, y^2) \overset{\text{def}}{=} \mathsf{succ}(x,y)$

Although producing two copies is common, triplication is likewise attested (Blust 2001), especially in sign langauges (Wilbur 2005). Triplication is likewise MSO-definable. The main modification is that we now use a copy set of size 3. In fact, a function which generates n of copies is MSO-definable as long the number of copies n is pre-determined. This is a crucial property for the computation reduplicative morphology. If the function cannot predetermine the number of copies, i.e., we need to calculate and count how many copies to generate, then the function is not MSO-definable.

2.4 Interim summary

The table below summarizes the generative capacity of reduplication and other morpho-phonological phenomena. We do not provide the type of logic needed to model the different string langauges.

	String Language	Grammar	String Function	Grammar	Logic
Most phonology	Regular Language	1-way FSA	Rational Function	1-way FST	Order-Preserving MSO
Most morphology	Regular Language	1-way FSA	Rational Function	1-way FST	Order-Preserving MSO
Partial reduplication	Regular Language	1-way FSA	Rational Function	1-way FST	Order-Preserving MSO
Total reduplication	Multiple Context-Free Language	MCFG	Regular Function	2-way FST	Non-Order-Preserving MSO

Table 1: Generative capacity and grammars for different morpho-phonological phenomena

The takeaway from this section is that there is a definite upper bound on the generative capacity of reduplication. Specifically, total reduplication requires at most MSO logic. Even though the output can contain multiple copies of the input, the logical function directly generates these copies without counting (cf. the use of transformational rules in Aronoff 1976; Lieber 1980; Carrier 1979). With these computational properties in place, the question now is whether linguistic theories for reduplication are above or below this threshold. As we will show, the answer is unclear for most theories, but negative for some.

3 Previous work on computing theories of reduplication

There are a large number of theories that have been developed for reduplication. However, there is little work on determining the generative capacity of these theories. This section discusses why. We focus on problems in using OT-based solutions to reduplication. We briefly preview a less common theory, Precedence-Based Phonology.

Since Moravcsik (1978)'s seminar paper on reduplication typology, there have been many generative treatments of reduplication (Marantz 1982; Steriade 1988; McCarthy and Prince 1995; Raimy 2000a; Kiparsky 2010; Inkelas and Zoll 2005). There are many overviews of developments in reduplication typology and theory (Hurch 2005; Raimy 2011; Urbanczyk 2007, 2011; Inkelas and Downing 2015a,b). However, most of these theories do not have explicit algorithms or implemented software. Thus, most of them have unclear generative capacity. There are two exceptions to the above predicament, but each with its own problems. These are Base-Reduplicant Correspondence Theory and Precedence-Based Phonology.

Base-Reduplicant Correspondence Theory (BRCT, McCarthy and Prince 1995) is couched within Optimality Theory (OT, Prince and Smolensky 2004) and it is a popular approach. OT has some finite-state approximations; see footnote 2 and Hao (2019) for an overview. However, these approximations are explicitly **approximations**. They impose restrictions on the constraints and principles behind OT because OT can compute functions which are supra-regular or not finite-state definable. This makes it difficult to determine the exact generative capacity of OT on its own. In fact, there is evidence that OT is computationally intractable in the first place (Idsardi 2006), though with some caveats (Heinz et al. 2009; Kornai 2009).

In terms of reduplication, Albro (2000, 2003, 2005) develops an implementation of OT that uses MCFGs in order to encode base-reduplicant constraints. He uses MCFGs to generate a pair of reduplicated copies, and then intersects the MCFG with an FSM that handles the phonological constraints. The phonological constraints are defined in terms of Primitive OT, an approximation of OT that is finite-state definable (Eisner 1997). However, it is unclear what is the generative capacity of the entire system, whether in terms of recognizing or transforming totally-reduplicated strings.

The second theory that we discuss is Precedence-Based Phonology (PBP: Raimy 1999, 2000a,b; Papillon 2020). Unlike OT, PBP was developed within serial or derivational rule-based approaches to phonology. PBP works by modifying the immediate precedence relations among segments in order to generate reduplicated strings. It has inspired a handful of other derivational theories of reduplication (Harris and Halle 2005; Halle 2008; Reiss and Simpson 2009; Frampton 2009). PBP is commonly adopted in modern serial systems of phonology and morphology, especially in combination with Search-and-Copy approaches to vowel harmony (Samuels 2010, 2011; Shen 2016).

Because PBP doesn't rely on parallelism, optimization, or an infinite candidate set, PBP is easier to implement and to computationally evaluate. Raimy (1999) develops the most explicit formalization of PBP. He likewise specifies a simple data structure and algorithm for computing the representations used in PBP. Because of this explicitness, we use PBP as a our main case study. We show that as a reduplication theory, the generative capacity of PBP is higher than the generative capacity of reduplication. The reason is because PBP relies on counting the number of reduplicative 'instructions' ($\sim$ back-loops) in order to determine how many copies to generate. This means that reduplication is MSO-definable, while PBP is not.

4 Precedence-Based Phonology

Having shown the difficulty in evaluating theories of reduplication, this section discusses Precedence-Based Phonology (PBP) in depth. As a formal system, PBP is explicit enough in its principles and procedures that we can develop an exact computational model for it. The next section does that. The current section explains the theory.

4.1 Immediate Precedence in Precedence-Based Phonology

Precedence-Based Phonology is a theory of representation. It argues for enriched phonological and phonetic representations which explicitly showcase the precedence relationships between segments. There are two types of precedence: immediate precedence and general precedence. A segment x immediately precedes y if x, y are adjacent and x is to left of y ($\ldots xy \ldots$). In contrast, a segment x generally precedes y if x is to the left of y ($\ldots x \ldots y \ldots$). General precedence is the transitive closure of immediate precedence. We illustrate the immediate precedences ($\lhd$) for a word *kæt* 'cat' in Figure 2, as a graph (2a) and set of arcs (2b). The symbols $\rtimes, \ltimes$ mark the start- and end-boundaries.[7]

Figure 2: Representing *kæt* with explicit immediate precedence

(a) Immediate precedence via graphs

$$\begin{tikzcd}
\rtimes \arrow[r, "\lhd"] & k \arrow[r, "\lhd"] & \text{æ} \arrow[r, "\lhd"] & t \arrow[r, "\lhd"] & \ltimes
\end{tikzcd}$$

(b) Immediate precedence via sets

$$\{\, (\rtimes \xrightarrow{\lhd} k), (k \xrightarrow{\lhd} \text{æ}), (\text{æ} \xrightarrow{\lhd} t), (t \xrightarrow{\lhd} \ltimes) \,\}$$

In mainstream phonology, a segment can immediately precede at most one segment, both in underlying representations (URs) and surface representations (SRs). PBP departs from this convention for URs. In the UR, individual morphemes form a single linear chain of immediate precedence[8]. But when two morphemes are combined, a node (segment or word-boundary) can end up immediately preceding multiple nodes. This symbol becomes a point of ambiguity.[9]

Figure 3: Immediate precedence for the UR of *kæt-s* 'cats'

(a) Non-linear graph for UR

$$\begin{tikzcd}
\rtimes \arrow[r, "\lhd"] & k \arrow[r, "\lhd"] & \text{æ} \arrow[r, "\lhd"] & t \arrow[r, "\lhd"] \arrow[d, "\lhd"'] & \ltimes \\
 & & & s \arrow[ur, "\lhd"'] &
\end{tikzcd}$$

(b) Non-linear set for UR

$$\{\, (\rtimes \xrightarrow{\lhd} k), (k \xrightarrow{\lhd} \text{æ}), (\text{æ} \xrightarrow{\lhd} t), (t \xrightarrow{\lhd} \ltimes), (t \xrightarrow{\lhd} s), (s \xrightarrow{\lhd} \ltimes) \,\}$$

Morphological operations create points of ambiguity, i.e., suffixation causes the final segment to immediately precede both the suffix and end-boundary $\ltimes$. Similarly, prefixation causes the start-boundary $\rtimes$ to immediately precede the prefix and the root. For example in the UR of *cats*, the segment *t* precedes multiple nodes: the end marker $\ltimes (t \xrightarrow{\lhd} \ltimes)$ and the suffix *s* $(t \xrightarrow{\lhd} s)$. When traversing the string from start $\rtimes$ to end $\ltimes$, there are now two paths. Although the morphology is allowed to create non-linear URs, the SR cannot contain any ambiguous points. The UR must turn into a flat linear string from $\rtimes$ to $\ltimes$ such that a symbol can immediately precede at most one other symbol. To illustrate, we show the immediate precedences for the SR of *kæts* 'cats'.

[7]The nature of precedence relations in PBP is further elaborated in (Raimy 2003). We notationally depart from Raimy (1999) who marks the edge-boundaries with #, %.

[8]See Papillon (2020) for discussion of morphemes that are not strictly linear

[9]Positions which can get targeted for multiple immediate precedences are usually psycholinguistically salient, e.g., the initial/final segment, leftmost/rightmost vowel. These positions are called pivot points or anchor points in the PBP literature (Fitzpatrick 2004; Raimy 2009b; Samuels 2010).

Figure 4: Immediate precedence for the SR of *kæt-s* ‘cats’

(a) Linear graph for SR

$$\begin{tikzcd} \rtimes \arrow[r, "\lhd"] & k \arrow[r, "\lhd"] & \text{æ} \arrow[r, "\lhd"] & t \arrow[r, "\lhd"] & s \arrow[r, "\lhd"] & \ltimes \end{tikzcd}$$

(b) Linear set for SR

$$\{ (\rtimes \xrightarrow{\lhd} k), (k \xrightarrow{\lhd} \text{æ}), (\text{æ} \xrightarrow{\lhd} t), (t \xrightarrow{\lhd} s), (s \xrightarrow{\lhd} \ltimes) \}$$

By allowing URs to contain multiple paths, PBP provides a creative solution for representing reduplication. In PBP, reduplication is modeled by creating a back-loop edge from the final segment to the initial segment. This back-loop is the reduplicative morpheme itself. Intuitively, it is an instruction to reduplicate. For example, given a base *kæt* ‘cat’, its reduplicated form *kæt*∼*kæt* ‘cat∼cat’ has an edge from *t* to *k*. When this non-linear UR is converted to a linear SR, we traverse the input form ⋊ to *t*, back to *k*, towards *t* again, and then end at ⋉. This linearization creates multiple copies of the input segments, e.g., k_1,k_2.

Figure 5: Immediate precedence for /kæt^2/→[kæt∼kæt] ‘cat∼cat’

(a) Non-linear graph for UR

$$\begin{tikzcd} \rtimes \arrow[r] & k \arrow[r] & \text{æ} \arrow[r] & t \arrow[r] \arrow[ll, bend right] & \ltimes \end{tikzcd}$$

(b) Non-linear set for UR

$$\{ (\rtimes \xrightarrow{\lhd} k), (k \xrightarrow{\lhd} \text{æ}), (\text{æ} \xrightarrow{\lhd} t), (t \xrightarrow{\lhd} \ltimes), (t \xrightarrow{\lhd} k) \}$$

(c) Linear graph for SR

$$\begin{tikzcd} \rtimes \arrow[r] & k_1 \arrow[r] & \text{æ}_1 \arrow[r] & t_1 \arrow[dll] & \\ & k_2 \arrow[r] & \text{æ}_2 \arrow[r] & t_2 \arrow[r] & \ltimes \end{tikzcd}$$

(d) Linear set for SR

$$\{ (\rtimes \xrightarrow{\lhd} k_1), (k_1 \xrightarrow{\lhd} \text{æ}_1), (\text{æ}_1 \xrightarrow{\lhd} t_1), (t_1 \xrightarrow{\lhd} k_2), (k_2 \xrightarrow{\lhd} \text{æ})_2, (\text{æ}_2 \xrightarrow{\lhd} t)_2, (t_2 \xrightarrow{\lhd} \ltimes) \}$$

PBP was originally developed to model reduplication (Raimy 1999, 2000a; Harrison and Raimy 2004). But by allowing URs where input symbols can immediately precede multiple symbols, PBP provides intuitive graphical representations for diverse morphological phenomena (cf. Downing 2001). These phenomena include gemination, insertion, deletion (Raimy and Cairns 2011), infixation, truncation (Raimy 2000a), multiple reduplication (Fitzpatrick 2006; Fitzpatrick and Nevins 2004), templatic morphology (Raimy 2007), prosodic templates (Raimy 2009b,a), diachronic change in reduplication (**??**), and ludlings or language games (Idsardi and Raimy 2005; Guimarães and Nevins 2012). Papillon (2020) develops an extensive coverage of PBP for such areas as allomorphy, tone, vowel harmony, and other processes. We provide a sketch below on how different processes can be encoded with non-linear paths.

Figure 6: Schema of morphological processes with PBP

$$\begin{tikzcd} p \arrow[dr] & & i \arrow[d] & & & & & & & s \arrow[d] \\ \rtimes \arrow[u, "\text{prefix}"] \arrow[r] & 1 \arrow[r] \arrow[ur, "\text{infix}"] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] \arrow[ll, bend right, "\text{reduplication}"'] & 5 \arrow[r] \arrow[loop above, "\text{gemination}"] & 6 \arrow[r] \arrow[rr, bend left, "\text{truncation}"] & 7 \arrow[r] & 8 \arrow[r] \arrow[ur, "\text{suffix}"] & \ltimes \end{tikzcd}$$

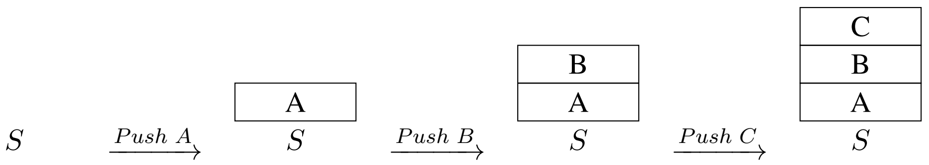
4.2 Stacks as data structures for immediate precedence

PBP can model a rich set of morphological information by having non-linear paths in the UR. However, these non-linear URs must be converted to linear SRs in a process of *serialization* or *linearization*. Before discussing algorithms for this conversion, this section discusses appropriate data structures for encoding the multiple immediate precedences of PBP's URs.

Although graphs are visually easy to read, they can be tricky to compute. A strand of work in PBP has focused on finding appropriate algorithms and data structures for non-linear URs (Fitzpatrick and Nevins 2002, 2004; Nevins 2004; Coleman 2004; Idsardi and Shorey 2007; McClory and Raimy 2007; Guimarães and Nevins 2012; Papillon 2020). The earliest proposal is Raimy (1999) who encodes immediate precedence relations via stacks. Because all subsequent work develops off of Raimy (1999), we focus on Raimy (1999) as our case study. We translate his data structures and algorithm into MSO logic.

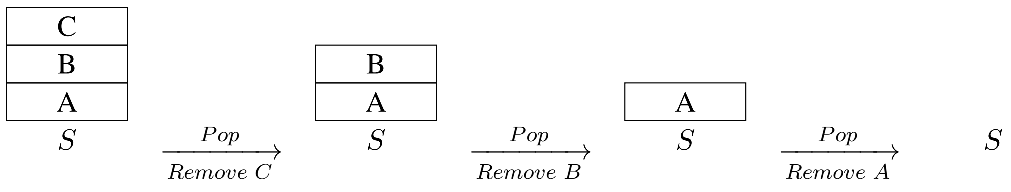
A stack is a data structure that acts as a list of elements which are ordered in terms of when they were inserted. Stacks have a LIFO (Last-In First-Out) structure such that only the last element added (pushed) can be removed (popped). To illustrate, let S be a stack and let A, B, C be three objects. We can individually push or add the objects A, B, C to the stack in order to get $S = [A, B, C]$.

Figure 7: Example of continuous pushing onto a stack



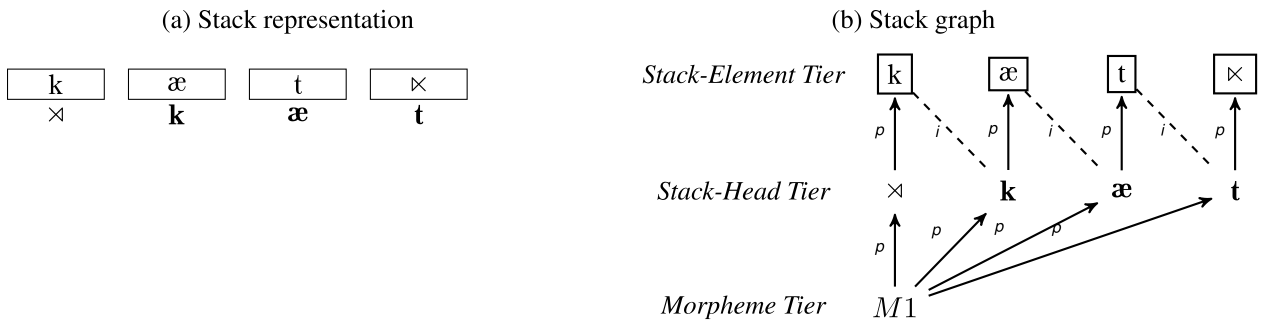
If we wanted to pop the stack to remove an object, then we have access to only the topmost element. Given the stack $S = [A, B, C]$, we can continuously pop the stack to remove C, B, and finally A.

Figure 8: Example of continuous popping from a stack



Raimy (1999) uses stacks to encode the multiple immediate precedence relations for an individual input symbol. For example in *kæt* 'cat', the start-boundary ⋊ and every segment (in **bold**) act as **stack-heads**. They point to a stack which keeps track of their immediate precedences. We translate these stack representations into an MSO graph that we call a *stack graph*. We later apply MSO logical transductions to these stack graphs.

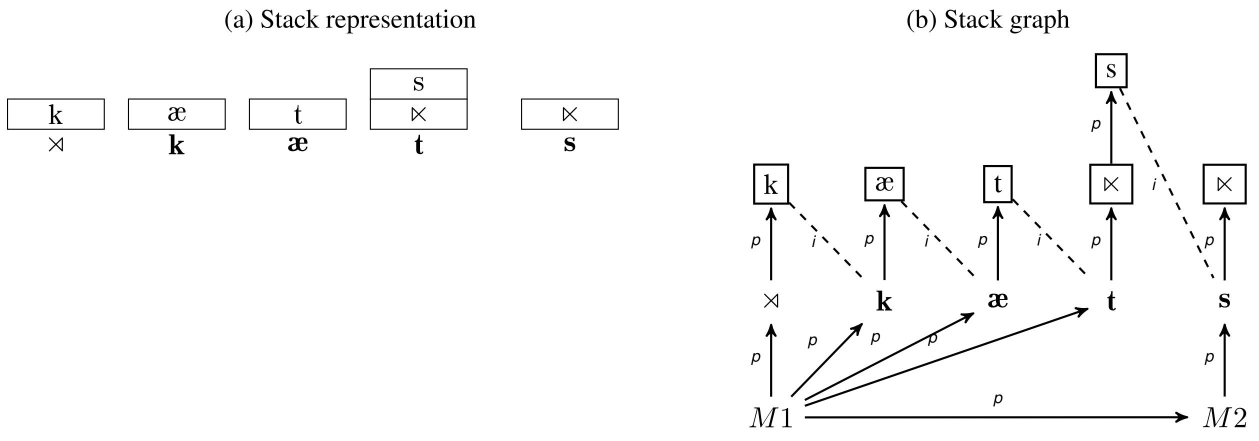
Figure 9: Stack representation and graph for /kæt/ ‘cat’



The above graph consists of three layers or tiers: a morpheme tier, stack-head tier, and stack-element tier. These tiers keep track of all the morphemes, segments, and precedence in the input. The morpheme *points* to the nodes that it contains via the *p*-labeled arcs. These nodes are on the stack-head tier. Each of them points to its precedence stack, specifically to the bottom element of its stack.[10] The stack elements are in boxes. They are indexed or identified with one or more of the input symbols on the stack-head tier via *i*-labeled edges (in dashes).

In the above stack representations for *kæt* ‘cat’, all the stacks contain a single element. This is because the input is monomorphemic. For a complex word like *kæt-s* ‘cats’, the stack for *t* contains two elements. At the bottom of the *t*-stack we have $\ltimes$, and at the top we have the suffix *s*. The bottom element $\ltimes$ points to the newly added element *s*. This signifies that the immediate precedence $t\xrightarrow{\lhd}\ltimes$ was added before $t\xrightarrow{\lhd}s$. We omit the names of the tiers.

Figure 10: Stack representation and graph for /kæt/ ‘cat’

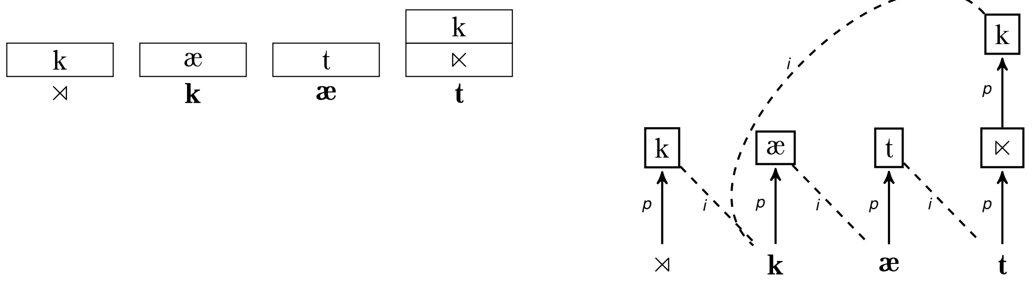


For the stack representation, the wider space between the *t*-stack and *s*-stack represents the fact they belong to two different morphemes. As for the stack graph, the morpheme tier explicitly includes an additional morpheme. The root morpheme *M1* points to the new suffix morpheme *M2*. In this way, the tiers keep track of the relative recency of the different immediate precedences and morphemes.

[10]Although it is counter-intuitive, the stack-head points to the bottom of the stack instead of to the top of the stack. We do this to make the graphs easier to read. This does not make the stacks first-in-first-out.

To further illustrate the model, we show the stack representation and stack graph for the reduplicated word *kæt∼kæt*. We omit the morpheme tier because there is only one morpheme with overt segments.

Figure 11: Stack representation and stack graph for *kæt∼kæt*



(a) Input stack representation (b) Input stack graph

5 Linearization

The previous section explained the basics of Precedence-Based Phonology (PBP) and how PBP laxes constraints on immediate precedence in URs. This section discusses linearization, the process whereby non-linear URs are converted to linear URs.[11] In general, linearization requires two actions: 1) deleting sub-optimal edges of immediate precedence, and 2) repeating segments that will be copied. The optimality of edge-deletion is elaborated in Raimy (2000a) and Idsardi and Raimy (2008).

As a case study, we formalize the conversion algorithm in Raimy (1999:ch4.4) or R99. R99 consists of three steps: *Compaction*, *End-Check*, and *Calling*. A brief summary of all three can be found in Raimy (1999:ch3.3). We first define some preliminary predicates for linearization (§5.1). We then formalize Compaction (§5.2) and Calling (§5.3). We do not formalize End-Check because it is restricted to cases of simple affixation.[12] Given the representation of the input and output of R99, we show that the linearization of reduplication is not MSO definable. The problem is that the algorithm's iterativity indirectly performs counting.

5.1 Preliminaries

We define a simple set of user-defined predicates in order to examine our stack graphs. The user-defined predicate $\mathbf{top\_of\_stack}(x)$ checks if a stack-element x is at the top of its stack. Analogously, the predicate $\mathbf{bottom\_of\_stack}(x)$ checks if x is at the bottom of its stack.

(3) *User-defined predicates for stack location*

a. $\mathbf{top\_of\_stack}(x) \overset{\text{def}}{=} \mathsf{stack\_elem}(x) \wedge \neg\exists y[\mathsf{stack\_elem}(y) \wedge \mathsf{point}(x,y)]$

b. $\mathbf{bottom\_of\_stack}(x) \overset{\text{def}}{=} \mathsf{stack\_elem}(x) \wedge \neg\exists y[\mathsf{stack\_elem}(y) \wedge \mathsf{point}(y,x)]$

[11]This process is called *linearization* in early work in PBP (Raimy 1999), while more recent work uses the term *serialization* (Idsardi and Raimy 2013). Because we use the term *linearization*. We follow Raimy (1999).

[12]The role of End-Check is to find the optimal word-initial and word-final segments when the input has prefixes or suffixes. It is MSO-definable as long as the input includes a morpheme tier.

Some parts of R99's algorithm must reference long-distance relationships between two nodes. For example, let x point to y, and and y point to z. By the transitive closure of 'pointing', x non-immediately points to z. The transitive closure of 'pointing' is formalized by the predicate $\mathbf{closed\_point}(X)$. It picks out the set of nodes X which can be reached from some given node x. Specifically, if x is in X, and x points to y, then y must be part of X. With this transitive closure, long-distance or non-immediate pointing is defined by $\mathbf{non\_imm\_point}(x, y)$. A node x non-immediately points to y if 1) y is on a path of transitive pointing from x, and 2) x, y are different nodes.

(4) *User-defined predicates for non-immediate pointing*

a. $\mathbf{closed\_point}(X) \overset{\text{def}}{=} \forall x, y[(x \in X \wedge \mathsf{point}(x, y)) \rightarrow y \in X]$

b. $\mathbf{non\_imm\_point}(x, y) \overset{\text{def}}{=} \forall X[(x \in X \wedge \mathbf{closed\_point}(X)) \rightarrow y \in X] \wedge x \neq y$

With non-immediate pointing defined, we can check whether a stack-element x is part of a stack-head y's stack. The following predicate does this by checking if y non-immediately points to x.

(5) *User-defined predicate for finding a stack-element in a stack*

$\mathbf{in\_stack\_of}(x, y) \overset{\text{def}}{=} \mathsf{stack\_elem}(x) \wedge \mathsf{stack\_head}(y) \wedge \mathbf{non\_imm\_point}(y, x)$

5.2 Compaction: finding unambiguous precedences

The first step in R99's linearization algorithm is Compaction. In this stage, we find areas in the UR that are already linear. To illustrate, consider the reduplicated word *kæt∼kæt* 'cat∼cat'. The UR consists of a single line from ⋊ to ⋉, and a back-loop from *t* to *k*. Because of this back-loop, the segment *t* is a point of ambiguity: it immediately precedes multiple symbols. At the segment *t*, we can traverse the graph in one of two directions: $t \xrightarrow{\lhd} k$ vs. $t \xrightarrow{\lhd} ⋉$. These ambiguous lines are dashed in red. All other lines (solid, in black) form a linear chain from one node to another.

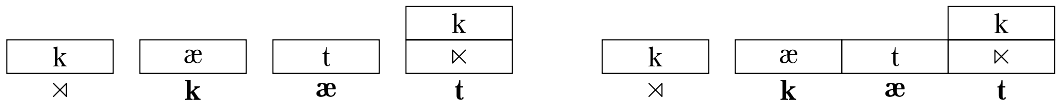
Figure 12: Ambiguous vs. un-ambiguous paths for *kæt∼kæt*

$$\begin{tikzcd}
⋊ \arrow[r] & \text{k} \arrow[r] & \text{æ} \arrow[r] & \text{t} \arrow[r, dashed] \arrow[ll, dashed, bend right] & ⋉
\end{tikzcd}$$

Compaction finds the above unambiguous linear paths between segments, i.e., between stack-heads. In terms of stack representation, two stack-heads x, y are compacted into a single block of stacks x:y if 1) x, y are segments, 2) the x-stack contains only one stack element y' such that 3) y, y' are indexed as the same segment. The ⋊-stack cannot be compacted to another stack because ⋊ is not a segment.[13]

Figure 13: Compacting stack representations for *kæt∼kæt*

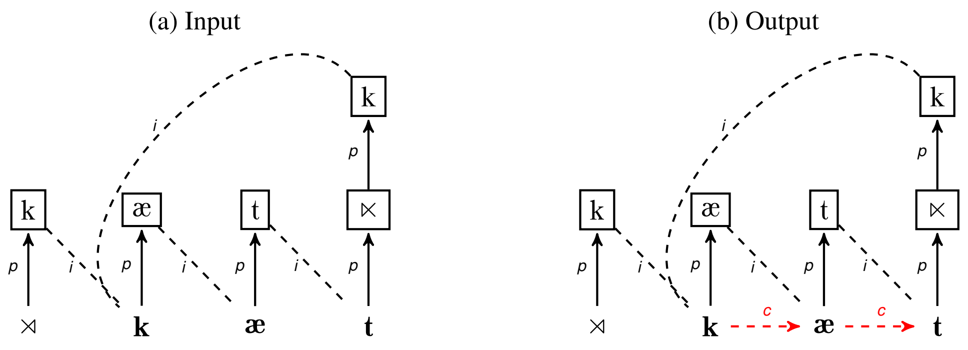
(a) Input stack representation (b) Compacted stack representation



[13]The stack size of y has no bearing on whether or not x's stack can be compacted with y's (Raimy 1999:185-6).

Compaction is an MSO-definable process. It is a function with a copy set of size 1. We show the input and output stack graphs below. In R99, Compaction is represented by simply reshuffling the stacks such that the compacted stacks are part of one contiguous block. We represent the property of being 'compacted' with c-labeled edges (dashed in red).

Figure 14: Input and output stack graphs for compacting *kæt*∼*kæt*



R99 treats compaction as an iterative process that compacts two stacks at a time. However, this iteration isn't needed. The logical formalization can apply compaction in a single step via the output function $\phi\texttt{compact}(x, y)$. This function will compact two stack-heads x, y if 1) x, y are segments, 2) there exists a stack-element z on the bottom of the x-stack, 3) z is simultaneously at the top of the x-stack, and 4) z, y are indexed as the same segment.

(6) *Output function for Compaction*
$\phi\texttt{compact}(x^1, y^1) \overset{\mathsf{def}}{=} \mathsf{stack\_head}(x) \wedge \mathsf{stack\_head}(y) \wedge \mathsf{segment}(x) \wedge \mathsf{segment}(y) \wedge$
$\exists z[\mathsf{stack\_elem}(z) \wedge \mathsf{point}(x, z) \wedge \textbf{top\_of\_stack}(z) \wedge \mathsf{indexed}(z, y)]$

Later stages of linearization will use non-immediate or long-distance version of the *compact* relation. We first define the transitive closure of *compact* relations, and use it to define non-immediate compacting.

(7) *User-defined predicates for non-immediate compacting*
a. $\textbf{closed\_compact}(X) \overset{\mathsf{def}}{=} \forall x, y[(x \in X \wedge \mathsf{compact}(x, y)) \rightarrow y \in X]$
b. $\textbf{non\_imm\_compact}(x, y) \overset{\mathsf{def}}{=} \forall X[(x \in X \wedge \textbf{closed\_compact}(X)) \rightarrow y \in X] \wedge x \neq y$

5.3 Calling: Iteratively projecting precedences

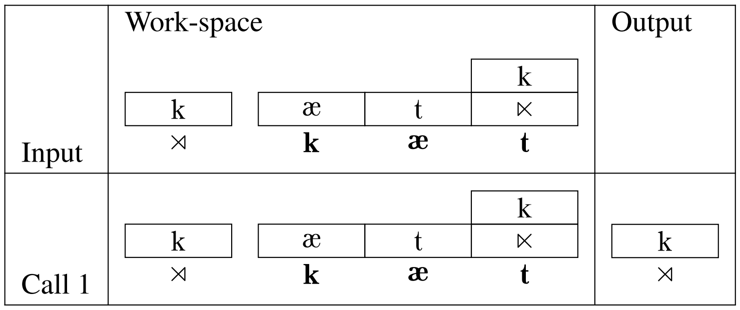
The final stage of R99 is Calling wherein we traverse the UR and generate a linear string. Unlike Compaction, Calling requires iteration and it references properties of both its input and interim output. There is no bound on the size of the copy set for Calling. This makes Calling non-MSO-definable. We first illustrate Calling with the simple example of 'cat∼cat' (§5.3.1). We formalize an MSO approximation of Calling that uses a copy set of size 3 (§5.3.2). We then show how this approximation is not feasible for generating more copies (§5.3.3).

5.3.1 Illustration of Calling

Calling is the final stage whereby we traverse the graph and stack representation in order to flatten our multi-linear UR to a linear SR. Calling is broken down into an unbounded sequence of calls. Each call consists of projecting or outputting a stack, finding its topmost element, and then deleting this topmost element if it is not at the bottom of the stack. Subsequent calls start projecting the stack of the previous call's last stack-element. We do this until we reach the $\ltimes$ symbol.

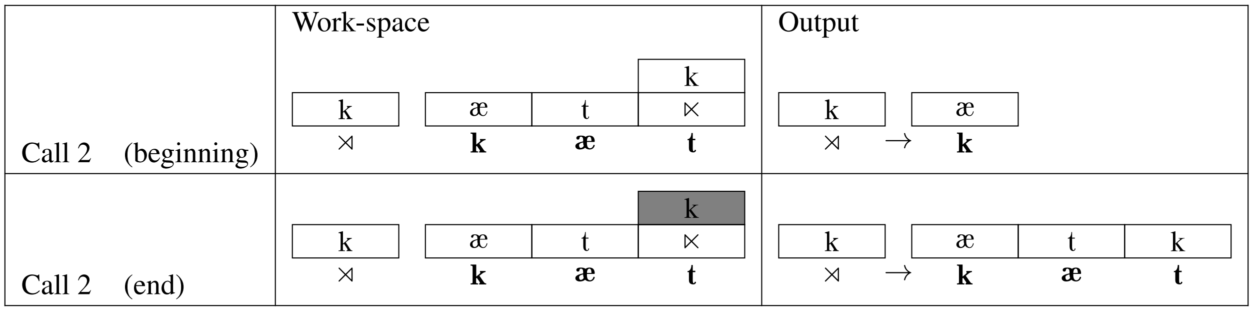
This mechanism is better explained by example. Consider the stack representation for 'cat∼cat'. The input simultaneously acts as our work-space which Calling can regularly modify. We start by examining the $\rtimes$-stack. We project the stack-head $\rtimes$ and its topmost stack-element: *k*. We re-examine the input/work-space. We check if the stack-element *k* was at the bottom of the $\rtimes$-stack in the work-space. If not, then we delete it from our work-space; otherwise, we do nothing. This constitutes Call 1.

Figure 15: Calling *kæt∼kæt*



In Call 2, we examine the output and find the last stack ($\rtimes$-stack) its stack-element (*k*). We then check the input and find the stack-head which *k* is indexed with: the *k*-stack. We project the stack-head *k* and its topmost stack-element *ae*. We connect the $\rtimes$-stack and *k*-stack via immediate precedence. As before, we try to modify the input/work-space by deleting the stack-element *æ* if it's not at the bottom of the stack in the input. It is at the bottom, so we delete nothing.

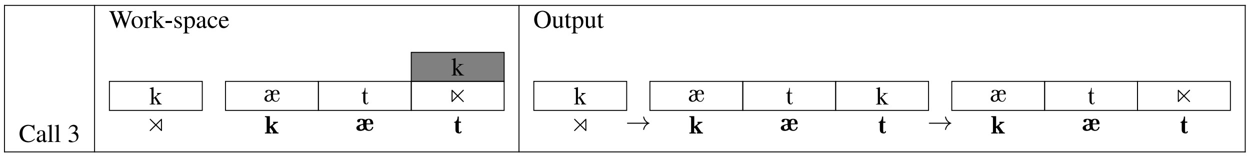
Figure 16: Calling *kæt∼kæt*



Thanks to Compaction, we likewise project any stack-heads which the *k*-stack was compacted into. In this case, the last stack that we project is the *t*-stack. We likewise project its topmost stack-element: *k*. We examine the work-space and check if the stack-element *k* is at bottom of the stack. It is not, so we **delete** it from the work-space. This ends Call 2.

Calling continuously applies until we see the ⋉ symbol. Since we haven't seen it yet, we apply Call 3. Call 3 repeats the same actions as Call 2. We find the last stack from Call 2: the *t*-stack. We find its projected stack-element *k*. We then project a *copy* of its indexed stack, the *k*-stack, along with any compacted stacks. Here, the final stack in Call 3 is a new copy of the *t*-stack. We project its topmost stack-element: ⋉. We do not project the stack-element *k* because it was deleted at the end of Call 2. At the end of Call 3, we check if the last stack-element in Call 3 is the ⋉ symbol. Because it is, Calling is finished and we output the linear SR: *kæt∼kæt.* Linearization ends.

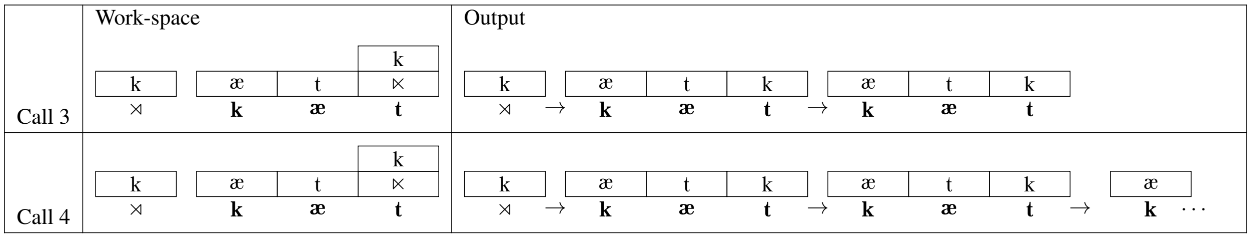
Figure 17: Calling *kæt∼kæt*



The critical step for linearization *kæt∼kæt* is at the end of Call 2. At this step, we project *k* as a stack-element, and then delete it from the work-space. When Call 3 starts, the algorithm will examine the projected stack-element *k* and output a new copy of *kæt.* The last stack in Call 3 is the new *t*-stack. Because Call 2 deleted the stack-element *k* from the work-space, Call 3 will project ⋉ as the stack-element for the *t*-stack.

To better understand this point, consider what would happen if Call 2 did not delete the stack-element *k*. Call 3 would output the same segments *kæt.* But in the work-space, the *t*-stack still has *k* at the top of its stack. Thus Call 3 would project *k* as the stack-element for the *t*-stack. This would cause the algorithm to start another Call 4, which would output a third copy of the base: *kæt∼kæt∼kæt*. . .. In fact, if we don't delete the stack element *k* at all, then the algorithm will not end.

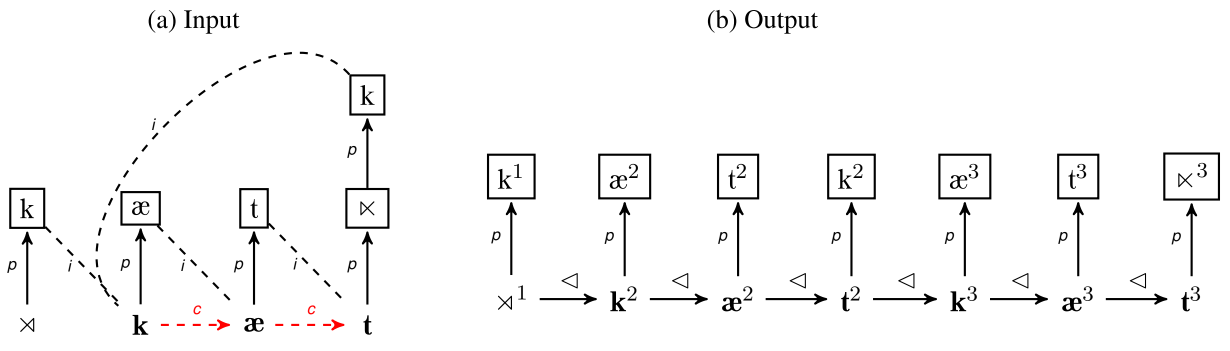
Figure 18: Calling *kæt∼kæt*



5.3.2 Formalization

R99's Calling stage is inherently iterative because it requires modifying its input as a work-space. It regularly modifies it in order to output a linearized reduplicated string. Our MSO formalization cannot do so. We develop an approximation of Calling that can handle simple reduplication. It is a function with a copy set of size 3, one for each Call. We show the input and output stack graphs below. Each call/copy outputs a specific set of nodes and links them via immediate precedence or successor $\lhd$. The output nodes have a superscript that marks what copy they belong to.

Figure 19: Input and output stack graphs for Calling *kæt∼kæt*



In Call 1 (Copy 1), we output the ⋊-stack and its topmost element *k*. We let ⋊ node point to its *k* node. Although we don't show it, all the labels on these two nodes are faithfully outputted.

(8) *Output functions to apply Call 1 over the ⋊-stack*

a. $\phi\rtimes(x^1) \overset{\text{def}}{=} \rtimes(x)$

b. $\phi\texttt{slack\_elem}(x^1) \overset{\text{def}}{=} \mathsf{slack\_elem}(x) \wedge \textbf{top\_of\_stack}(x) \wedge \exists y[\rtimes(y) \wedge \textbf{in\_stack\_of}(x,y)]$

c. $\phi\texttt{point}(x^1,y^1) \overset{\text{def}}{=} \phi\rtimes(x^1) \wedge \phi\texttt{slack\_elem}(y^1)$

Although vacuous for 'cat∼cat', we check if the topmost stack-element on the ⋊-stack is at the bottom of the stack. If it is, then we 'delete' it from our work-space. Our logical formalization is declarative so we cannot delete anything in our input, nor can we output and then delete an element. As an approximation, we define a user-defined predicate that essentially marks some stack-element x as invisible for any operation that follows Call 1 (after Copy 1). A stack-element x is marked as invisible if it is 1) the input stack-element which got outputted in Copy 1, and 2) not the bottom element in the input stack.

(9) *User-defined predicate to potentially 'delete' the last stack-element in Call 1*
$\textbf{invisible:Call}\textit{1}(x) \overset{\text{def}}{=} \mathsf{stack\_elem}(x) \wedge \phi\texttt{slack\_elem}(x^1) \wedge \neg\textbf{bottom\_of\_stack}(x)$

After Call 1 has projected the ⋊-stack, the subsequent calls will generate the various segments and their successor relations. In fact, Calls 2 and Calls 3 perform virtually the same set of operations in order to generate the two copies of *kæt*. Their behavior can be generalized into a set of schemata. Each schema is parameterized by a value *n* which is interpreted as the current Call or the designated Copy.[14]

At the beginning of a Call *n* (Call 2), we find the last stack-head which was projected in the previous Call (⋊-stack). We find its projected stack element (*k*). We then find the stack-head which is indexed with this stack-element (the *k*-stack).

(10) *Schemata for finding previous call's last stack-head, its stack-element, and the element's indexed stack-head*

[14]This does *not* mean that the transduction can actually count nor does it treat the copies as actual integers. We are treating them as simple natural numbers for readability.

a. $\textbf{final\_stack\_head\_in\_prevCall}\textit{n-1}(x) \overset{\text{def}}{=} \phi\texttt{stack\_head}(x^{n-1}) \wedge \neg\exists y[\phi\texttt{stack\_head}(y^{n-1}) \wedge \phi\texttt{succ}(x^{n-1}, y^{n-1})]$

b. $\textbf{final\_stack\_elem\_in\_prevCall}\textit{n-1}(x) \overset{\text{def}}{=} \phi\texttt{stack\_elem}(x^{n-1}) \wedge \exists y[\phi\texttt{stack\_head}(y^{n-1}) \wedge \textbf{in\_stack\_of}(x, y) \wedge \textbf{final\_stack\_head\_in\_prevCall}\textit{n-1}(y)]$

c. $\textbf{Indexed\_with\_Call}\textit{n-1}(x) \overset{\text{def}}{=} \mathsf{stack\_head}(x) \wedge \exists y[\textbf{final\_stack\_head\_in\_prevCall}\textit{n-1}(y) \wedge \mathsf{indexed}(y, x)]$

We output this stack-head (*k*) along with all the stack-heads that it is compacted into (*æt*). All these stack-heads are connected via immediate precedence, from the last stack from the previous Copy (Copy 1, $\rtimes$-stack) to the last stack in current Copy (Copy 2, *t*-stack).

(11) *Schemata for projecting and connecting the stack-heads of the current call*

a. $\phi\texttt{stack\_head}(x^n) \overset{\text{def}}{=} \mathsf{stack\_head}(x) \wedge [\textbf{Indexed\_with\_Call}\textit{n-1}(x) \vee \exists y[\mathsf{stack\_head}(y) \wedge \textbf{Indexed\_with\_Call}\textit{n-1}(y) \wedge \textbf{non\_imm\_compact}(y, x)]]$

b. $\phi\texttt{succ}(x^{n-1}, y^n) \overset{\text{def}}{=} \textbf{final\_stack\_head\_in\_prevCall}\textit{n-1}(x^{n-1}) \wedge \phi\texttt{stack\_head}(y^n) \wedge \textbf{Indexed\_with\_Call}\textit{n-1}(y)$

c. $\phi\texttt{succ}(x^n, y^n) \overset{\text{def}}{=} \phi\texttt{stack\_head}(x^n) \wedge \phi\texttt{stack\_head}(y^n) \wedge \mathsf{compact}(x, y)$

For each of the projected stack-heads, we project the highest stack-element which is *not* marked as invisible by the previous Calls (*æ, t, k*). For Call 2, this extra condition is superfluous because the previous Call 1 worked on only the $\rtimes$-stack. For later Calls, this restriction is critical. The projected stack-heads must point to their projected stack-elements.

(12) *Schemata for projecting and connecting the stack-elements of the current call*

a. $\phi\texttt{stack\_elem}(x^n) \overset{\text{def}}{=} \mathsf{stack\_elem}(x) \wedge \exists y[\mathsf{stack\_head}(y) \wedge \textbf{in\_stack\_of}(x, y) \wedge \phi\texttt{stack\_head}(y^n) \wedge \forall z[(\mathsf{stack\_elem}(z) \wedge \textbf{in\_stack\_of}(z, y) \wedge \textbf{non\_imm\_point}(x, z)) \rightarrow (\textbf{Invisible:Call1}(z) \vee \ldots \vee \textbf{Invisible:Call}\textit{n-1}(x)]]$

b. $\phi\texttt{point}(x^n, y^n) \overset{\text{def}}{=} \phi\texttt{stack\_head}(x^n) \wedge \phi\texttt{stack\_elem}(y^n) \wedge \textbf{in\_stack\_of}(x, y)$

Finally, the current Call will examine the projected stack-elements and potentially 'delete' them from the input. If a projected stack-element is not at the bottom of its stack in the input, then it is made invisible. For Call 2 over *kæt∼kæt*, this predicate will mark the stack-element *k* on the *t*-stack as invisible for future Calls.

(13) *Schema for potentially 'deleting' the last stack-element in the current call*

$\textbf{invisible:Call}\textit{n} \overset{\text{def}}{=} \mathsf{stack\_elem}(x) \wedge \phi\texttt{stack\_elem}(x^n) \wedge \neg\textbf{bottom\_of\_stack}(x)$

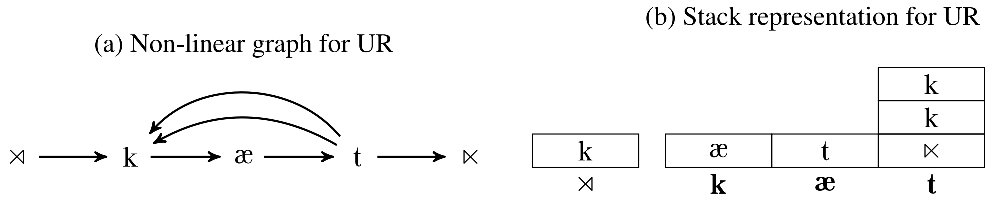
To generate *kæt∼kæt*, we need 3 calls so the above schemata are instantiated for each of Call 2 and Call 3. This requires replacing the parameter *n* with 2,3 for each of the schemata, e.g., $\phi\texttt{stack\_head}(x^2)$, $\phi\texttt{stack\_head}(x^3)$, etc. These schemata are capable of marking all the right stack-elements as invisible for each corresponding call. For Call 2, the predicate $\textbf{Invisible:Call}\textit{1}(x)$ blocks the 'deletion' of the stack-element *k* on the $\rtimes$-stack. But for Call 3, $\textbf{Invisible:Call}\textit{2}(x)$ marks the stack-element *k* on the *t*-stack. This causes Call 3 to project the stack-element $\ltimes$ for the *t*-stack.

5.3.3 Scaling to triplication

The previous section defined a declarative and logical formalization for R99's linearization algorithm. However, the formalization is an inexact approximation. It cannot scale up to triplication where the input is repeated three times: *kæt∼kæt∼kæt.* This is because the above function has a copy set of size 3. It can only do 3 calls, whereas R99 requires using 4 calls.

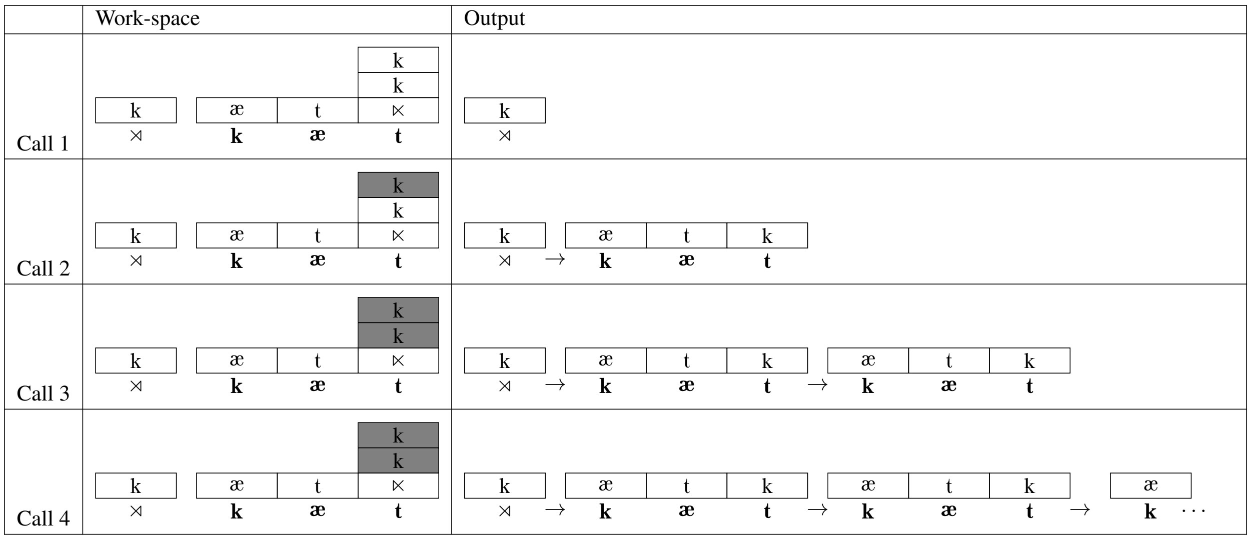
For R99, the input to triplication would include two back-loops from the final segment *t* to the first segment *k*. We show the input UR and stack representation.

Figure 20: Immediate precedence for triplication: /kæt^3/→[kæt∼kæt∼kæt] 'cat∼cat'



R99 requires four calls where each of the last three calls would generate a new copy. Calls 2 and 3 would each delete a copy of the stack-element *k* from the *t*-stack.

Figure 21: Calling *kæt∼kæt∼kæt*



For our declarative formalization, we need to add an additional Copy to the copy set. We can use the previous section's schemata in order to define the right set of predicates and output functions for Call 4 where *n* is 4, e.g. ϕ`stack_head`(x^4), and so forth.

However, in order to treat R99 as a general-purpose reduplication algorithm, we would need to instantiate the schemata for every possible call, e.g., 2 calls for simple reduplication, 3 calls for triplication, 4 for quadruplication, and so forth. Each call is triggered by the presence of back-loop which essentially acts as an instruction to reduplicate. R99 assumes that there is no bound on how many back-loops can be added to the input. This means that there is no bound on the number of need calls nor on the number of copies in the copy set.

Mathematically, if a logical transduction does not have a fixed copy set, then it is not MSO. This fact is stated as Fact 1.37 in Courcelle and Engelfriet (2012). In the appendix, we show a formal proof that Calling is not MSO-definable because it does not have a fixed copy set.

6 Discussion and Conclusion

This chapter established the logical expressivity needed for reduplication. Although reduplication is computationally more complex than most morphological and phonological processes, it has an explicit and definable generative capacity. Specifically, reduplication is MSO-definable. In contrast, most theories of reduplication are under-formalized and inexact. This makes it difficult to determine if these theories are computationally sufficient and restrictive enough to handle reduplication.

As a case study, we formalize one such theory, Precedence-Based Phonology (PBP). We showed that reduplication is MSO-definable while PBP is not. Thus, PBP is too powerful. Behind the formulas, we argue that the main reason why PBP is not MSO is because PBP utilizes counting. It needs to count the number of reduplicative instructions in the input (backloops). This causes PBP to simulate a type of iterative grammar that can rewrite its own input. However, we did not define what the exact power or generative capacity of PBP. We only established that PBP is beyond MSO-definability.

In future work, we aim to evaluate PBP using other computational tools. We conjecture that PBP can be simulated with pebble-based transducers (Engelfriet and Maneth 2002; Engelfriet 2015). We likewise conjecture that nearly all theories of reduplication utilize counting to some extent and are thus also not MSO-definable. Thus, this chapter is further evidence for the need for formal exactness, not just in theoretical formulation but in explicit and well-documented implementations and algorithms.

A Formal proof that Calling is not MSO-definable

To formally prove that Raimy (1999)'s linearization algorithm is not MSO-definable, we will use the following fact about MSO transductions. Specifically, MSO transductions must have a copy set of a fixed size (Courcelle 1997:31). This is stated as Fact 1.37 in Courcelle and Engelfriet (2012):

(14) Fact 1.37 from Courcelle and Engelfriet (2012)
"For every monadic second-order transduction f there exists an integer k such that, if f transforms a relational structure U into a relational structure S, then $|S| \leq k|U|$."

The variable $|U|$ is the size of the input, $|S|$ is the size of the output, and k is the size of the copy set. What follows is first an illustration of possible values of k needed for different sizes of input. We do this to determine the specific variables involved in determining the size of both the input and output. We then use these variables to show the simple proof that Calling is not MSO-definable.

Taking reduplication of 'cat^{x}' /kæt^{x}/ as the input to our Calling transduction, let's assume that our transduction is MSO-definable. There is then a positive integer k such that k is the size of our copy set. For any inputs of size $|U|$, the size of the output should be $|S| < k|U|$. Ignoring Compaction and Call 1, for *cat*^{1}, the variables are evaluated as follows.

(15) *Applying Calling for* cat^{1}

a. Let k be a fixed number
b. In the input: $|U| = 5 + 1 = 6$, where
   i. 5 is the number of segment-heads and stack-elements besides the % symbol, and
   ii. 1 also is the number of back-loops stack elements and the % symbol.
   iii. 1 essentially acts as the number of times *x* which the stem *kæt* appears.
c. In the output: $|S| = 6$
d. Combining sizes of the input and copy set:
   $k|U| = 6k$
e. Applying Fact 1.37:
   If $|S| \leq k|U|$, then $6 \leq 6k$
f. k must have a value such that k$\geq$ 1

If we do not specify the number of times the stem must be reduplicated, then our input is *cat*^{x} where the variables are further refined as follows.

(16) *Applying Calling for* cat^{x}

a. Let k be a fixed number
b. In the input: $|U| = 5 + x$
c. In the output: $|S| = 6x$
d. Combining sizes of the input and copy set:
   $k|U| = k(5 + x)$
e. Applying Fact 1.37:
   If $|S| \leq k|U|$, then $6x \leq k(5 + x)$
f. k must have a value such that $k \geq 6x/(5 + x)$

If we increase the stem size, such as for a hypothetical input *badcat*^{x}, then we further refine the variables as follows.

(17) *Applying Calling for a longer stem* badcat^{1}

a. Let k be a fixed number
b. In the input: $|U| = 11 + x$
c. In the output: $|S| = 12x$
d. Combining sizes of the input and copy set:
   $k|U| = k(11 + x)$

e. Applying Fact 1.37:
   If $|S| \leq k|U|$, then $12x \leq k(11+x)$

f. k must have a value such that $k \geq 12x/(11+x)$

Abstracting away from all this, for any input *g* we can establish some of these variables as follows.

(18) *Applying Calling for any input g*

a. Let k be a fixed number

b. In the input: $|U| = y + x$, where
   i. *y* is the number of segments and stack elements besides the % symbol, and
   ii. *x* is the number of times the stem appears.
   iii. The value *x* includes the number of back-looping stack elements and the % symbol

c. In the output: $|S| = (y+1)x$

d. Combining sizes of the input and copy set:
   $k|U| = k|U| = k(y+x)$

Given this general template for Calling, we now show that Calling is not MSO-definable because it cannot be the case that $|S| \leq k|U|$.

(19) *General template for evaluating Calling on an input g*

a. Let $y = x = 3k$

b. Applying Fact 1.37:
   $|S| \leq k|U|$
   $(y+1)x \leq k(x+y)$
   $(3k+1)(3k) \leq k(3k+3k)$
   $9k^2 + 3k \leq 6k^2$
   $9k^2 \leq 6k^2$
   This is false. Therefore, there is no fixed positive integer *k* such that $|S| \leq k|U|$. The algorithm is not MSO-definable.

In §5.3, we provided an MSO transduction for a restricted version of Calling. For that transduction, k is a fixed integer. However, for that transduction to work, the values of x and/or y have to be bounded by some fixed number z so that the transduction can be MSO-definable, i.e. that the number of calls can fit within our fixed copy set. Restricting the size of x would mean that not all logically possible sizes of stems can be linearized; while restricting y would mean not all logically possible number of reduplications are possible.

References

Albro, D. M. (2000). Taking Primitive Optimality Theory beyond the finite state. In J. Eisner, L. Karttunen, and A. Thériault (Eds.), *Finite-state phonology: Proceedings of the 5th Workshop of SIGPHON*, Luxembourg, pp. 57–67.

Albro, D. M. (2003). A large-scale, computerized phonological analysis of malagasy. In *Talk presented at the Annual Meeting of the Linguistic Society of America*.

Albro, D. M. (2005). *Studies in Computational Optimality Theory, with Special Reference to the Phonological System of Malagasy*. Ph. D. thesis, University of California, Los Angeles, Los Angeles.

Aronoff, M. (1976). *Word formation in generative grammar*. Number 1 in Linguistic Inquiry Monographs. Cambridge, MA: The MIT Press.

Beesley, K. and L. Karttunen (2000). Finite-state non-concatenative morphotactics. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, ACL '00, Hong Kong, pp. 191–198. Association for Computational Linguistics.

Beesley, K. and L. Karttunen (2003). *Finite-state morphology: Xerox tools and techniques*. Stanford, CA: CSLI Publications.

Blust, R. A. (2001). Thao triplication. *Oceanic Linguistics 40*(2), 324–335.

Büchi, J. R. (1960). Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly 6*(1-6), 66–92.

Carrier, J. L. (1979). *The interaction of morphological and phonological rules in Tagalog: a study in the relationship between rule components in grammar.* Ph. D. thesis, Massachusetts Institute of Technology.

Chandlee, J. (2014). *Strictly Local Phonological Processes*. Ph. D. thesis, University of Delaware, Newark, DE.

Chandlee, J. (2017). Computational locality in morphological maps. *Morphology 27*(4), 1–43.

Chandlee, J. and J. Heinz (2012). Bounded copying is subsequential: Implications for metathesis and reduplication. In *Proceedings of the 12th Meeting of the ACL Special Interest Group on Computational Morphology and Phonology*, SIGMORPHON '12, Montreal, Canada, pp. 42–51. Association for Computational Linguistics.

Chandlee, J. and A. Jardine (2019). Quantifier-free least fixed point functions for phonology. In *Proceedings of the 16th Meeting on the Mathematics of Language (MoL 16)*, Toronto, Canada. Association for Computational Linguistics.

Clark, A. (2017). Computational learning of syntax. *Annual Review of Linguistics 3*, 107–123.

Clark, A. and R. Yoshinaka (2012). Beyond semilinearity: Distributional learning of parallel multiple context-free grammars. In *International Conference on Grammatical Inference*, pp. 84–96.

Clark, A. and R. Yoshinaka (2014). Distributional learning of parallel multiple context-free grammars. *Machine Learning 96*(1-2), 5–31.

Cohen-Sygal, Y. and S. Wintner (2006). Finite-state registered automata for non-concatenative morphology. *Computational Linguistics 32*(1), 49–82.

Cohn, A. C. (1989). Stress in Indonesian and bracketing paradoxes. *Natural language & linguistic theory 7*(2), 167–216.

Coleman, J. (2004). Thesis shmesis: Representing reduplication with directed graphs. Bachelor's thesis, Haverford College, Haverford, PA.

Courcelle, B. (1997). The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformations*, Volume 1, pp. 313–400. World Scientific.

Courcelle, B. and J. Engelfriet (2012). *Graph Structure and Monadic Second-Order Logic, a Language Theoretic Approach*. Cambridge: Cambridge University Press.

Culy, C. (1985). The complexity of the vocabulary of Bambara. *Linguistics and Philosophy 8*, 345–351.

Danis, N. and A. Jardine (2019). Q-theory representations are logically equivalent to autosegmental representations. In *Proceedings of the Society for Computation in Linguistics*, Volume 2, pp. 29–38.

Dolatian, H. (2020). *Computational locality of cyclic phonology in Armenian*. Ph. D. thesis, Stony Brook University.

Dolatian, H. and J. Heinz (2018). Modeling reduplication with 2-way finite-state transducers. In *Proceedings of the 15th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, Brussells, Belgium. Association for Computational Linguistics.

Dolatian, H. and J. Heinz (2020). Computing and classifying reduplication with 2-way finite-state transducers. *Journal of Language Modeling 8*, 79–250.

Downing, L. J. (2001). Review of eric raimy (2000). the phonology and morphology of reduplication.(studies in generative grammar 52.) berlin: Mouton de gruyter. pp. viii+ 200. *Phonology 18*(3), 445.

Eisner, J. (1997). Efficient generation in primitive optimality theory. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 313–320.

Eisner, J. (2000a). Directional constraint evaluation in optimality theory. In *COLING 2000 Volume 1: The 18th International Conference on Computational Linguistics*.

Eisner, J. (2000b). Easy and hard constraint ranking in ot: Algorithms and complexity. In *Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*, pp. 22–33.

Engelfriet, J. (2015). Two-way pebble transducers for partial functions and their composition. *Acta Informatica 52*(7-8), 559–571.

Engelfriet, J. and H. J. Hoogeboom (2001, April). MSO definable string transductions and two-way finite-state transducers. *Transactions of the Association for Computational Linguistics 2*(2), 216–254.

Engelfriet, J. and S. Maneth (2002). Two-way finite state transducers with nested pebbles. In *International Symposium on Mathematical Foundations of Computer Science*, pp. 234–244. Springer.

Filiot, E. and P.-A. Reynier (2016, August). Transducers, logic and algebra for functions of finite words. *ACM SIGLOG News 3*(3), 4–19.

Fitzpatrick, J. (2004). A concatenative theory of possible affix types. In A. Salanova (Ed.), *Papers from EVELIN I. MIT Working Papers in Linguistics*.

Fitzpatrick, J. (2006). Sources of multiple reduplication in Salish and beyond. In S. T. Bischoff, L. Butler, P. Norquest, and D. Siddiqi (Eds.), *MIT Working Papers on Endangered and Less Familiar Languages: Studies in Salishan 7*, pp. 211–240.

Fitzpatrick, J. and A. Nevins (2002). Phonological occurrences: Relations and copying. In *Proceedings of the 2nd North American Phonology Conference*, Montreal.

Fitzpatrick, J. and A. Nevins (2004). Linearizing nested and overlapping precedence in multiple reduplication. In *University of Pennsylvania Working Papers in Linguistics*, pp. 75–88.

Frampton, J. (2009). *Distributed reduplication*. Cambridge, MA: MIT Press.

Frank, R. and G. Satta (1998). Optimality theory and the generative complexity of constraint violability. *Computational linguistics 24*(2), 307–315.

Gazdar, G. and G. K. Pullum (1985). Computationally relevant properties of natural languages and their grammars. *New generation computing 3*, 273–306.

Gerdemann, D. and G. Van Noord (2000). Approximation and exactness in finite state optimality theory. In *Procedings of the 5th Meeting of the ACL Special Interest Group in Computational Phonology*, pp. 34–45.

Graf, T. (2010a). Comparing incomparable frameworks: A model theoretic approach to phonology. In *University of Pennsylvania Working Papers in Linguistics*, Volume 16.

Graf, T. (2010b). Logics of phonological reasoning. Master's thesis, University of California, Los Angeles.

Guimarães, M. and A. Nevins (2012). Opaque nasalization in ludlings and the precedence relations of reduplication and infixation. *Letras & Letras 28*(1), 129–166.

Halle, M. (2008). Reduplication. *Current studies in linguistics series 45*, 325.

Hao, Y. (2019). Finite-state optimality theory: non-rationality of harmonic serialism. *Journal of Language Modelling 7*(2), 49–99.

Harris, J. and M. Halle (2005). Unexpected plural inflections in Spanish: Reduplication and metathesis. *Linguistic Inquiry 36*(2), 195–222.

Harrison, K. D. and E. Raimy (2004). Reduplication in tuvan: Exponence, readjustment and phonology. In *Proceedings of Workshop in Altaic Formal Linguistics*, Volume 1. Citeseer.

Heinz, J. and W. Idsardi (2013). What complexity differences reveal about domains in language. *Topics in Cognitive Science 5*(1), 111–131.

Heinz, J., G. M. Kobele, and J. Riggle (2009). Evaluating the complexity of optimality theory. *Linguistic Inquiry 40*(2), 277–288.

Hopcroft, J. E. and J. D. Ullman (1969). *Formal languages and their relation to automata*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Hulden, M. (2009). *Finite-state machine construction methods and algorithms for phonology and morphology*. Ph. D. thesis, University of Arizona.

Hulden, M. and S. T. Bischoff (2009). A simple formalism for capturing reduplication in finite-state morphology. In J. Piskorski, B. Watson, and A. Yli-Jyrä (Eds.), *Proceedings of the 2009 conference on Finite-State Methods and Natural Language Processing: Post-proceedings of the 7th International Workshop FSMNLP 2008*, Amsterdam, pp. 207–214. IOS Press.

Hurch, B. (Ed.) (2005). *Studies on reduplication*. Number 28 in Empirical Approaches to Language Typology. Berlin: Walter de Gruyter.

Idsardi, W. and E. Raimy (2008). Reduplicative economy. In B. Vaux and A. Nevins (Eds.), *Rules, Constraints, and Phonological Phenomena*, Chapter 5, pp. 149–184. Oxford: Oxford University Press.

Idsardi, W. and E. Raimy (2013). Three types of linearization and the temporal aspects of speech. In E. Raimy and C. Cairns (Eds.), *Challenges to linearization*, pp. 31–56. Mouton de Gruyter, Berlin.

Idsardi, W. J. (2006). A simple proof that optimality theory is computationally intractable. *Linguistic Inquiry 37*(2), 271–275.

Idsardi, W. J. and E. Raimy (2005). Remarks on language play. Unpublished manuscript, University of Delaware, Newark DE.

Idsardi, W. J. and R. Shorey (2007). Unwinding morphology. Presented at CUNY Phonology Forum Workshop on Precedence Relations.

Inkelas, S. and L. J. Downing (2015a). What is reduplication?? T typology and analysis part 1/2: The typology of reduplication. *Language and Linguistics Compass 9*(12), 502–515.

Inkelas, S. and L. J. Downing (2015b). What is reduplication? Typology and analysis part 2/2: The analysis of reduplication. *Language and Linguistics Compass 9*(12), 516–528.

Inkelas, S. and C. Zoll (2005). *Reduplication: Doubling in Morphology*. Cambridge: Cambridge University Press.

Jardine, A., N. Danis, and L. Iacoponi (2020). A formal investigation of Q-theory in comparison to autosegmental representations. *Linguistic Inquiry*.

Johnson, C. D. (1972). *Formal aspects of phonological description*. The Hague: Mouton.

Kaplan, R. M. and M. Kay (1994). Regular models of phonological rule systems. *Computational linguistics 20*(3), 331–378.

Karttunen, L. (1998). The proper treatment of optimality in computational phonology. *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, 1–12.

Karttunen, L. (2003). Computing with realizational morphology. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pp. 203–214. Springer.

Kiparsky, P. (2010). Reduplication in stratal OT. *Reality exploration and discovery: Pattern interaction in language & life*, 125–142.

Kobele, G. M. (2006). *Generating Copies: An investigation into structural identity in language and grammar*. Ph. D. thesis, University of California, Los Angeles.

Kornai, A. (2009). The complexity of phonology. *Linguistic Inquiry 40*(4), 701–712.

Koskenniemi, K. (1983). *Two-level morphology: A General Computational Model for Word-Form Recognition and Production*. Ph. D. thesis, University of Helsinki.

Lieber, R. (1980). *On the organization of the lexicon*. Ph. D. thesis, Massachusetts Institute of Technology.

Manaster-Ramer, A. (1986). Copying in natural languages, context-freeness, and queue grammars. In *Proceedings of the 24th annual meeting on Association for Computational Linguistics*, pp. 85–89. Association for Computational Linguistics.

Marantz, A. (1982). Re reduplication. *Linguistic Inquiry 13*(3), 435–482.

McCarthy, J. J. and A. Prince (1995). Faithfulness and reduplicative identity. In J. N. Beckman, L. W. Dickey, and S. Urbanczyk (Eds.), *Papers in Optimality Theory*. Amherst, MA: Graduate Linguistic Student Association, University of Massachusetts.

McClory, D. and E. Raimy (2007). Enhanced edges: morphological influence on linearization.

Miller, P. H. (1999). *Strong generative capacity: The semantics of linguistic formalism*. Stanford: CSLI publications.

Moravcsik, E. (1978). Reduplicative constructions. In J. Greenberg (Ed.), *Universals of Human Language*, Volume 1, pp. 297–334. Stanford, California: Stanford University Press.

Nevins, A. (2004). What ug can and can't do to help the reduplication learner. In A. Cslrmaz, A. Gualmini, and A. Nevins (Eds.), *MIT Working Papers in Linguistics 48*, pp. 113–126. Cambridge,MA: MIT Department of Linguistics and Philosophy.

Oakden, C. (2020). Notational equivalence in tonal geometry. *Phonology 37*, 257–296.

Papillon, M. (2020). *Precedence and the lack thereof: Precedence-relation-oriented phonology*. Ph. D. thesis, University of Maryland.

Payne, A., M. H. Vu, and J. Heinz (2017). A formal analysis of correspondence theory. In *Proceedings of the Annual Meetings on Phonology*, Volume 4.

Prince, A. and P. Smolensky (2004). *Optimality Theory: Constraint Interaction in Generative Grammar*. Oxford: Blackwell Publishing.

Raimy, E. (1999). *Representing reduplication*. Ph. D. thesis, University of Delaware, Newark, DE.

Raimy, E. (2000a). *The Phonology and Morphology of Reduplication*. Berlin: Mouton de Gruyter.

Raimy, E. (2000b). Remarks on backcopying. *Linguistic Inquiry 31*(3), 541–552.

Raimy, E. (2003). Asymmetry and linearization in phonology. In A. M. Di Sciullo (Ed.), *Asymmetry in grammar*, Volume 2, pp. 129–146. John Benjamins Publishing.

Raimy, E. (2007). Precedence theory, root and template morphology, priming effects and the structure of the lexicon.

Raimy, E. (2009a). A case of appendicitis. See Raimy and Cairns (2009), pp. 177–188.

Raimy, E. (2009b). Deriving reduplicative templates in a modular fashion. See Raimy and Cairns (2009), pp. 383–404.

Raimy, E. (2011). Reduplication. See van Oostendorp et al. (2011), pp. 2383–2413.

Raimy, E. and C. Cairns (2011). Precedence relations in phonology. See van Oostendorp et al. (2011), pp. 799–823.

Raimy, E. and C. E. Cairns (Eds.) (2009). Number 48 in Current Studies in Linguistics. Cambridge, MA: MIT Press.

Reiss, C. and M. Simpson (2009). Reduplication as projection. Unpublished manuscript, Concordia University, Montréal.

Riggle, J. A. (2004). *Generation, recognition, and learning in finite state Optimality Theory*. Ph. D. thesis, University of California, Los Angeles.

Ritchie, G. (1989). On the generative power of two-level morphological rules. In *Proceedings of the fourth conference on European chapter of the Association for Computational Linguistics*, pp. 51–57. Association for Computational Linguistics.

Ritchie, G. (1992). Languages generated by two-level morphological rules. *Computational Linguistics 18*(1), 41–59.

Roark, B. and R. Sproat (2007). *Computational Approaches to Morphology and Syntax*. Oxford: Oxford University Press.

Rogers, J., J. Heinz, M. Fero, J. Hurst, D. Lambert, and S. Wibel (2013). Cognitive and sub-regular complexity. In G. Morrill and M.-J. Nederhof (Eds.), *Formal Grammar*, Volume 8036 of *Lecture Notes in Computer Science*, pp. 90–108. Springer.

Rogers, J. and G. Pullum (2011). Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information 20*, 329–342.

Samuels, B. (2010). The topology of infixation and reduplication. *The Linguistic Review 27*(2), 131–176.

Samuels, B. D. (2011). *Phonological architecture: A biolinguistic perspective*. Oxford Studies in Biolinguistics. Oxford University Press.

Savitch, W. J. (1982). *Abstract machines and grammars*. Boston: Little Brown and Company.

Savitch, W. J. (1989). A formal model for context-free languages augmented with reduplication. *Computational Linguistics 15*(4), 250–261.

Seki, H., T. Matsumura, M. Fujii, and T. Kasami (1991). On multiple context-free grammars. *Theoretical Computer Science 88*(2), 191–229.

Seki, H., R. Nakanishi, Y. Kaji, S. Ando, and T. Kasami (1993). Parallel multiple context-free grammars, finite-state translation systems, and polynomial-time recognizable subclasses of lexical-functional grammars. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pp. 130–139. Association for Computational Linguistics.

Shen, D. T.-C. (2016). *Precedence and search: Primitive concepts in morpho-phonology*. Ph. D. thesis, National Taiwan Normal University, Taipei, Taiwan.

Stabler, E. P. (2004). Varieties of crossing dependencies: structure dependence and mild context sensitivity. *Cognitive Science 28*(5), 699–720.

Steriade, D. (1988). Reduplication and syllable transfer in Sanskrit and elsewhere. *Phonology 5*(1), 73–155.

Strother-Garcia, K. (2018). Imdlawn Tashlhiyt Berber syllabification is quantifier-free. In *Proceedings of the Society for Computation in Linguistics*, Volume 1, pp. 145–153.

Strother-Garcia, K. (2019). *Using model theory in phonology: a novel characterization of syllable structure and syllabification*. Ph. D. thesis, University of Delaware.

Struijke, C. M. (2000). *Reduplication, feature displacement, and existential faithfulness*. Ph. D. thesis, University of Maryland, College Park.

Urbanczyk, S. (2007). Themes in phonology. In P. de Lacy (Ed.), *The Cambridge Handbook of Phonology*, pp. 473–493.

Urbanczyk, S. (2011). Reduplication. In M. Aronoff (Ed.), *Oxford Bibliography*.

van Oostendorp, M., C. Ewen, E. Hume, and K. Rice (Eds.) (2011). *The Blackwell companion to phonology*. Malden, MA: Wiley-Blackwell.

Walther, M. (2000). Finite-state reduplication in one-level prosodic morphology. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, NAACL 2000, Seattle, Washington, pp. 296–302. Association for Computational Linguistics.

Wilbur, R. B. (1973). *The phonology of reduplication*. Ph. D. thesis, University of Indiana, Bloomington, Indiana.

Wilbur, R. B. (2005). A reanalysis of reduplication in American Sign Language. See Hurch (2005), pp. 595–623.