Final Project Report: Transducers for phonological rules

Zixuan Wang, *University of California, Los Angeles*

Abstract

Phonology has long been an important branch of linguistics and phonological rule is essential to the study of phonology as a formal way of describing phonological process in various languages. In this project, we are concerned with a design of finite-state transducers for phonological rules and application of a computational approach to analyze the phonological rules of natural language. Combined linguistics study with computer science, we explore the possibility of building a model of phonological rule systems in Haskell, which can be used to efficiently evaluate surface forms of words in natural language.

1. Introduction

Computational linguistics, an interdisciplinary field of studying linguistics from a computational perspective, is aimed to solve linguistics questions in a computational approach. Due to the characteristics and limitations of computer, we need to use finite state automata, a computational model which can be implemented by computer, to generate natural language. With the help of computer programs, we are able to compare the results produced by the finite state machine with the predicted results. In this way, we can explore phonological rules using computer. Since Haskell is a statically typed functional programming language, it is good for pattern matching and our computational linguistics study. In this project, we choose Haskell as our programming language for the development of transducers and the evaluation of phonological rules.

We would build a computer program for some fundamental phonological rule analysis, which combines the use of finite-state transducer with phonological rules.

1.1. Finite-state transducer

A finite-state automaton is consisted of a finite set of states, a finite set of symbols, a starting state, a set of ending states, and a set of transitions. We are able to have a graphical representation for a particular finite-state automaton and generate various strings defined by the particular finite-state automaton. In addition to a finite-state automaton, a probabilistic finite-state automaton is applying functions to associate different values with the initial-state, the final-state, and the transitions. In this way, we can calculate the value of generating a specific string for a particular probabilistic finite-state automaton. There are many kinds of values that can be associated with for a probabilistic finite-state automaton. A finite-state transducer is such a model with each starting event, ending event, and transition event associated with some set of strings in some output alphabet. In this project, we design a finite-state transducer equipped with functions that associate set of strings with generating the string to get the surface form of a word from the underlying form of a word.

In this project, the alphabet we used includes all the English letters plus a question mark denoting the glottal stop and a number sign demoting the word boundary.

1.2. Phonological rules

A phonological rule is a way of expressing a phonological process, which can be used to produce the surface forms of words from the underlying forms of words. A phonological rule typically consists of a title, an underlying sound that is changes, an arrow representing change of the sound on the left to the sound on the right, the sound that the underlying sound changes to, a slash meaning the notation to the right is where the phonological rule is applied, the sound that precedes the one to be changed, the location of the sound that is going to be changed, and the sound follows the one to be changed. Thus, the form of a phonological rule is basically: A -> B / C_D. In the project, we use a data type of the form: type PhonologicalRule = (Char, Char, Maybe Char, Maybe Char), to represent a phonological rule, where the first Char is the character in the underlying form, the second Char is the character in the surface form, the third component is the character before the character to be changed and the fourth component is the character after the character to be changed. The third and fourth components are of Maybe Char because there are rules that only car about one side. Hence, they can be Just Char or Nothing to describe the left and right parts of the context requirements of a phonological rule.

In this project, we are concerned with the following six phonological rules:

1. Assimilation: n -> m / _ b (n becomes m when it is before b)

2. Devoicing: b -> p / _ # (b becomes p when it is at the end of a word)

3. Glottalization: t -> ? / _ n (t becomes the glottal stop “?” when it is before n)

4. Voicing: f -> v / i _ e (f becomes v when it is between i and e)

5. Initial Voicing: s -> z / # _ (s becomes z when it is at the beginning of a word)

6. Vowel Lowering: i -> e / l _ (i becomes e when it is after l)

The phonological rules we explore here are constructed by ourselves only for the use of this project. These rules are inspired by English while they are not actual English phonological rules. Similarly, the words we use are also constructed for the purpose of evaluating different surface forms as the real phonetic symbols are hard to represent by characters supported by Haskell.

2. Function Design

We develop the following functions to generate transducers for phonological rules and evaluate the underlying form of a word corresponding with the surface form of a word. The inspiration comes from the assignment and a related research paper.

2.1. Phonological rule transducers

We write a function to take a phonological rule of the type mentioned above and generate a finite-state transducer for the phonological rule. The graphical representation of the generated transducer is similar to the graphical representation drawn in the last page of the fifth assignment documentation. We use three states represented by integers as in the assignments: the zero state as the starting state with the empty string, all the three states are the ending state with one state with the target and the other two with empty strings. This is the same as the finite-state transducer in the assignment. Then we design transitions using the approach in the paper, concatenating all possible transitions together. Basically, we are building transitions between states just like the transitions drawn in the assignment: from the state zero to the state zero, from the state zero to the state one, from the state one to the state zero, from the state one to the state one, from the state one to the state two, from the state two to the state one, and from the state two to the state zero. The state one is used to handle the environment before the sound to be changed and the state two is used to handle the environment after the sound to be changed.

2.2. Phonological rule evaluation

The evaluation functions are mainly of two types, returning a list of potential surface forms or return a Boolean. We write two functions returning a list of potential surface forms, one for applying a single phonological rule and the other for applying a list of phonological rules. The former one takes a single phonological rule and an underlying form of a word as input and outputs a list of possible surface forms of the word. The latter one takes a list of phonological rules in order and an underlying form of a word to evaluate possible surface forms of the word. This function is mainly used to see if different orders of applying phonological rules would generate different surface forms. We are using a string list as the input type and return type to keep consistent with the output produced by the finite-state transducer in the assignment because the "val" function we write in the assignment to calculate the value of generating string produces a string list for the transducer in the assignment and we would also apply the "val" function for evaluation in this project. So, we are using a string list for an underlying form input of a word and return a string list as a surface form of a word. The other two functions returning Booleans are used to check if a predicted form matches the surface form produced by the phonological rules applying to the underlying form of a word. Again, the one is for a single phonological rule application and the other is for multiple phonological rules in order. We also write a helper function to help implement the recursion process in the main checking functions.

3. Examples

We will give some examples of our implementation of functions: (Since the transducers derived by the function is too long for each phonological rule, we will use the evaluation functions to explore the surface forms. The finite-state transducer for a phonological rule generated is very similar to the one in the assignment and readers can call them by using the command: phonologicalDerivation *theRule)*

1. the surface form of "#canb#" applying *assimilation*:

*singleRuleDerivation assimilation ["#canb#"]

["#camb#"] (n becomes m as it is before b)

*oneRuleCheck assimilation ["#canb#"] "#camb#"

True

*oneRuleCheck assimilation ["#canb#"] "#canb#"

False

2. the surface form of "#lamb#" applying *devoicing*:

*singleRuleDerivation devoicing ["#lamb#"]

["#lamp#"] (b becomes p as it is at the end of the word)

*oneRuleCheck devoicing ["#lamb#"] "#lamp#"

True

*oneRuleCheck devoicing ["#lamb#"] "#lamb#"

False

3. the surface form of "#gotn#" applying *glottalization*:

*singleRuleDerivation glottalization ["#gotn#"]

["#go?n#"] (t becomes ? as it is before n)

*oneRuleCheck glottalization [“#gotn#”] “#go?n#”

True

*oneRuleCheck glottalization [“#gotn#”] “#gotn#”

False

4. the surface form of “#knife#” applying *voicing*:

*singleRuleDerivation voicing [“#knife#”]

[“#knive#”] (f becomes v as it is between i and e)

5. the surface form of “#sagon#” applying *initial vocing*:

*singleRuleDerivation initialVoicing [“#sagon#”]

[“#zagon#”] (s becomes z as it is at the beginning of a word)

6. the surface form of “#list#” applying *vowel lowering*:

*singleRuleDerivation vowelLowering [“#list#”]

[“#lest#”] (i becomes e as it is after l)

7. the surface form of “#life#” applying *voicing* and *vowel lowering* in order:

*ruleOrderDerivation [voicing, vowelLowering] [“#life#”]

[“#leve#”] (applying voicing rule first, f becomes v as it is between i and e; then applying vowel lowering rule, i becomes e as it is after l)

*checkResults [voicing, vowelLowering] [“#life#”] “#leve#”

True

*checkResults [voicing, vowelLowering] [“#life#”] “#life#”

False

*checkResults [voicing, vowelLowering] [“#life#”] “#live#”

False

*checkResults [voicing, vowelLowering] [“#life#”] “#lefe#”

False

8. the surface form of “#life#” applying *vowel lowering* and *voicing* in order:

*ruleOrderDerivation [vowelLowering, voicing] [“#life#”]

[“#lefe#”] (applying vowel lowering rule first, i becomes e as it is after l, destroying the environment that could have triggered voicing rule)

*checkResults [vowelLowering, voicing] [“#life#”] “#lefe#”

True

9. the surface form of “#ratnb#” applying *assimilation*, *devoicing*, and *glottalization* in order:

*ruleOrderDerivation [assimilation, devoicing, glottalization] [“#ratnb#”]

[“#ratmp#”] (applying assimilation rule first, n becomes m as it is before b; then applying devoicing rule, b becomes p as it is at the end of the word; the environment that could have triggered glottalization rule being destroyed by the previous assimilation rule applied)

10. the surface form of “#ratnb#” applying *devoicing*, *glottalization, and assimilation* in order:

*ruleOrderDerivation [devoicing, glottalization, assimilation] [“#ratnb#”]

[“#ra?np#”] (applying devoicing rule first, b becomes p as it is at the end of the word; then applying glottalization rule, t becomes ? as it is before n; the environment that could have triggered assimilation rule being destroyed by the previous devoicing rule applied)

As we can see, we are using our transducer generation function combined with our evaluation function to explore surface forms of underlying forms of words. We not only produce the surface form of a word from a given underlying form but also check the produced surface form with the predicted surface form. We can check if the phonological rule is applied to the word and it is meaningful especially for the case with a list of phonological rules because we can determine the rule order by using our check function. We notice that an underlying form of word can produce different surface forms depending on the order of the phonological rules to be applied. This is critical for our study of feeding, counterfeeding, bleeding, and counterbleeding with respect to phonological rules. We can determine whether a particular rule creates or destroy the environment that could have triggered other rules with our developed computer program.

4. Thoughts on improvement

We have successfully built a finite-state transducer to evaluate a surface form when there is one character to be changed with optionally one character before the character to be changed and on optionally one character after the character to be changed. We introduce the number sign to denote the word boundary so that we can examine phonological rules concerned with the beginning and the ending of the word. If we introduce the space character to our alphabet, then we might be able to evaluate phonological rules like deletion and epenthesis. If we use a list of characters in the type of phonological rule, we might be able to evaluate phonological rules with larger context, looking into characters further in a word. In these ways, we can expand our range of phonological rules to be explored and extend our study of rule ordering.

5. Conclusion

In this project, we build a function to generate a transducer for a particular phonological rule and write various functions to apply the transducer to analyze word surface forms plus

rule ordering. We develop a computational tool for the analysis of phonological rule and the finite-state model is useful for us to study feeding, counterfeeding, bleeding, and counterbleeding of phonological rules. Overall, we mainly explore the possibility of rewriting context-sensitive phonological rules into a corresponding finite-state transducer, which is part of computational phonology.

Reference

[1] Tim Hunter. *Assignment05: Generalizations of FSAs*. [Online]

Available from: https://ccle.ucla.edu/course/view/19W-LING185A-1?section=5 [Accessed: March 22nd 2019].

[2] Ronald M. Kaplan. Martin Kay. *Regular Models of Phonological Rule Systems*. [Online] Available from: https://web.stanford.edu/~mjkay/Kaplan%26Kay.pdf [Accessed: March 22nd 2019].

[3] Hayes B. *Introductory Phonology*. Wiley-Blackwell. 2009.