Implementing logic-based approaches to phonology

Hossep Dolatian

Stony Brook University

September 22, 2022

Table of Contents

Explaining the title

- Title: *Implementing logic-based approaches to phonology*
- What we'll do:
  1. See what logical approaches *look* like
  2. Provide software support for running them
  3. Discuss general problems for future research and expansion
- URL to software and tutorial: `tinyurl.com/hossepBMRS`

Pen-and-paper linguistics

1. Phonologists like finding patterns and making rules for them:
   - Dataset: { ba, pa, aba, *apa }
   - Rule: p $\rightarrow$ b / a_a
2. After you make a rule on pen-and-paper, it's good to double-check the rule to make sure it's correct
3. How do you do that?
   - Manually, easy but risky
   - Computationally, more effortful but more reliable

Debugging Pen-and-paper linguistics

- To computationally debug a phonological analysis, you do the following:
  1. Convert pen-and-paper analysis (A1) to an explicit format (F)
  2. Compile the format F into a machine M
  3. Use the machine to double-check your analysis

Debugging Pen-and-paper linguistics

- To computationally debug a phonological analysis, you do the following:
  1. Convert pen-and-paper analysis (A1) to an explicit format (F)
  2. Compile the format F into a machine M
  3. Use the machine to double-check your analysis
- A conventional-ish way to do that is using finite-state machines:
  1. Manually convert your rule (p→b/a_a) into a hand-drawn FST
  2. Compile your FST on Pynini, HFST, or whatever
  3. Run the FST on your data
- For FST users, there's a LOT of work on all these areas:
  - Conventional formats: diagrams, transition lists, ATT formats
  - Reliable compilers: Pynini, HFST, a myriad of packages
  - Some of these compilers can do steps (1-3) simultaneously with little human intervention

Debugging Pen-and-paper linguistics

- To computationally debug a phonological analysis, you do the following:
  1. Convert pen-and-paper analysis (A1) to an explicit format (F)
  2. Compile the format F into a machine M
  3. Use the machine to double-check your analysis
- A conventional-ish way to do that is using finite-state machines:
  1. Manually convert your rule (p→b/a_a) into a hand-drawn FST
  2. Compile your FST on Pynini, HFST, or whatever
  3. Run the FST on your data
- For FST users, there's a LOT of work on all these areas:
  - Conventional formats: diagrams, transition lists, ATT formats
  - Reliable compilers: Pynini, HFST, a myriad of packages
  - Some of these compilers can do steps (1-3) simultaneously with little human intervention
- We'll explore doing debugging and conversion with logic-based approaches

Table of Contents

Example logical transduction

- FSTs are a common computational framework for formalizing rules
- FSTs are equivalent to logical transductions, which are also explicit
- Example transduction for $p \rightarrow b / a\_a$

Alphabet	Σ	$=$	{a,p,b}
Functions	F	$=$	*succ(x), pred(x)*
Predicates	$\mathbf{a\_a}(x)$	$\stackrel{\text{def}}{=}$	$\mathsf{a}(\mathsf{pred}(x)) \wedge \mathsf{a}(\mathsf{succ}(x))$
Output formula	$\phi\mathsf{a}(x^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{a}(x)$
	$\phi\mathsf{b}(x^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{b}(x) \vee (\mathsf{p}(x) \wedge \mathbf{a\_a}(x))$
	$\phi\mathsf{p}(x^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{p}(x) \wedge \neg\mathbf{a\_a}(x)$

Example logical transduction

- For this example transduction, an input string /papab/ becomes [pabab]

$$\text{Input:} \quad p_0 \xrightarrow[\triangleleft]{} a_0 \xrightarrow[\triangleleft]{} p_0 \xrightarrow[\triangleleft]{} a_0 \xrightarrow[\triangleleft]{} b_0$$

$$\text{Output:} \quad p_1 \xrightarrow[\triangleleft]{} a_1 \xrightarrow[\triangleleft]{} b_1 \xrightarrow[\triangleleft]{} a_1 \xrightarrow[\triangleleft]{} b_1$$

- Note the following details:
  - Input elements belong to a set of nodes indexed by 0
  - Output elements belong to a set of nodes indexed by 1
  - Successor relations are unchanged and order-preserving (postpone discussion)

Format for logical transductions

- We converted our pen-and-paper rule to a small set of logical statements, but can we check that it's correct?

Alphabet	Σ	$=$	$\{a,p,b\}$
Functions	F	$=$	$succ(x), pred(x)$
Predicates	$\mathbf{a\_a}(x)$	$\stackrel{\text{def}}{=}$	$\mathsf{a}(\mathsf{pred}(x)) \wedge \mathsf{a}(\mathsf{succ}(x))$
Output formula	$\phi\mathsf{a}(x^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{a}(x)$
	$\phi\mathsf{b}(x^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{b}(x) \vee (\mathsf{p}(x) \wedge \mathbf{a\_a}(x))$
	$\phi\mathsf{p}(x^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{p}(x) \wedge \neg\mathbf{a\_a}(x)$

- First step: make sure this format is explicit enough
- But it's not because there's a lot of details in fonts/colors/superscripts

$\rightarrow$ Need to turn this format into a more explicit one

Table of Contents

Format for logical transductions

- For FST formats, some formats are very explicit and machine-readable, e.g, an FST is a sequence of transitions:
  (input state, input label, output state, output label)

→ Can essentially write all of an FST's information into a simple text file

Format for logical transductions

- For FST formats, some formats are very explicit and machine-readable, e.g, an FST is a sequence of transitions:
  (input state, input label, output state, output label)

→ Can essentially write all of an FST's information into a simple text file

- For logical transductions to our knowledge, there isn't a pre-existing format that's machine-readable and 100% explicit
- Thankfully, Eric Meinhardt has been making one using YAML syntax with inspiration from the BMRS notation (Bhaskar et al., 2020)

Format for logical transductions

- Compare our logical statements to the text file **BetweenAVoicing.txt**

Alphabet	Σ	$=$	{a,p,b}
Functions	F	$=$	*succ(x), pred(x)*
Predicates	$\mathbf{a\_a}(\mathrm{x})$	$\stackrel{\text{def}}{=}$	$\mathsf{a}(\mathsf{pred}(x)) \wedge \mathsf{a}(\mathsf{succ}(x))$
Output formula	$\phi\mathsf{a}(\mathrm{x}^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{a}(\mathrm{x})$
	$\phi\mathsf{b}(\mathrm{x}^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{b}(x) \vee (\mathsf{p}(x) \wedge \mathbf{a\_a}(x))$
	$\phi\mathsf{p}(\mathrm{x}^1)$	$\stackrel{\text{def}}{=}$	$\mathsf{p}(x) \wedge \neg\mathbf{a\_a}(x)$

Format for logical transductions

- Compare our logical statements to the text file **BetweenAVoicing.txt**

Alphabet	Σ	$=$	$\{a,p,b\}$
Functions	F	$=$	$succ(x), pred(x)$
Predicates	$\mathbf{a\_a}(x)$	$\overset{\text{def}}{=}$	$\mathsf{a}(\mathsf{pred}(x)) \wedge \mathsf{a}(\mathsf{succ}(x))$
Output formula	$\phi\mathsf{a}(x^1)$	$\overset{\text{def}}{=}$	$\mathsf{a}(x)$
	$\phi\mathsf{b}(x^1)$	$\overset{\text{def}}{=}$	$\mathsf{b}(x) \vee (\mathsf{p}(x) \wedge \mathbf{a\_a}(x))$
	$\phi\mathsf{p}(x^1)$	$\overset{\text{def}}{=}$	$\mathsf{p}(x) \wedge \neg\mathbf{a\_a}(x)$

- Some differences:
  - Specify alphabets, copy set size, and word boundaries
  - Specify predicates vs. output functions
  - Clarify if a node (and its labels) belongs to input vs. a copy via underscore
  - No support for negation or Boolean operations
  - Logic uses just **IF CONDITION RESULT ELSE** statements

  $\rightarrow$ such formats simulate a BMRS system $=$ rational function

Table of Contents

Compiling a logical transduction

- Given an explicit format for a logical transduction, next step is to compile it
- To my knowledge, there isn't a pre-existing compiler for BMRS-style logical transductions
- I made my own with Python that...
    1. Takes a text file with YAML syntax
    2. Converts the file to a Pythonic translation
    3. Uses dynamic programming (memoization) to run the translation on an input word

Demo

- Everything can work via the command line
    1. Download the repo
    2. On the commandline or terminal, type...
       python3 bmrs.py run BetweenAVoicing.txt 'papab'
- The code will convert the **BetweenAVoicing.txt** file to a Python file **BetweenAVoicing.py** with all the logical info
- The file is run on the input word /papab/ to generate [pabab]

Demo

- Other commands you can run are...
  1. Convert the YAML file to Python
     python3 bmrs.py convert BetweenAVoicing.txt
     or
     python3 bmrs.py convert BetweenAVoicing.txt BetweenAVoicing.py
  2. Run the compiler for the word /papab/ using either the text file or Python file
     python3 bmrs.py run BetweenAVoicing.txt 'papab'
     or
     python3 bmrs.py run BetweenAVoicing.py 'papab'
- Conversion errors and running errors are logged
- You can see the input-output graphs as a TSV file (nicer to look on Excel)

Table of Contents

What you can do

- The YAML format can encode any BMRS-style system that generates a rational function
- The Python compiler can thus also run any rational function
- If you want to debug your pen-and-paper analysis or your logical statements, you can convert your analysis to a text file and then run it with the Python software
- Bugs will likely be found
- Repo includes example YAML files for inserting nodes, deleting nodes, and repeating nodes

What you can't do (everywhere yet)

- **Features on YAML:**
  - As of now, the YAML format cannot directly encode features or natural classes.
  - If you want to make a transduction for intervocalic voicing, you'll need to create long predicates that pick out every vowel and consonant
  - Hopefully this will be fixed soon.

What you can't do (everywhere yet)

- **Features on YAML**:
  - As of now, the YAML format cannot directly encode features or natural classes.
  - If you want to make a transduction for intervocalic voicing, you'll need to create long predicates that pick out every vowel and consonant
  - Hopefully this will be fixed soon.
- **Features on Python**
  - The Python translation and compiler however CAN do this.
  - So you'll need to manually create your own Python file and use features there.
  - Check out the file **IntervocalicVoicing.py** to illustrate

What you can't do (everywhere yet)

- **More expressive functions on YAML:**
  - The YAML format can encode any rational function
  - But it can't encode anything more expressive

What you can't do (everywhere yet)

- **More expressive functions on YAML:**
  - The YAML format can encode any rational function
  - But it can't encode anything more expressive
- **More expressive functions on Python**
  - The Python translation and compiler however can encode some non-rational but regular functions
  - You can write functions that concatenate two rational functions: **X**$\rightarrow$**XX** (=the **Copying.py** file)
  - You can write a non-rational function with user-specified succ/pred relations like **X**$\rightarrow$**XX**$^{\mathbf{r}}$ can encode a non-rational function with
  - But you can't write a regular function that needs quantifers to find succ/pred values. For example, a function *u1-u2-...-un*$\rightarrow$*un...u2u1* where *ux* is a substring

What you can't do (everywhere yet)

- **More expressive functions on YAML:**
  - The YAML format can encode any rational function
  - But it can't encode anything more expressive
- **More expressive functions on Python**
  - The Python translation and compiler however can encode some non-rational but regular functions
  - You can write functions that concatenate two rational functions: **X**$\rightarrow$**XX** (=the **Copying.py** file)
  - You can write a non-rational function with user-specified succ/pred relations like **X**$\rightarrow$**XX**$^{\mathbf{r}}$ can encode a non-rational function with
  - But you can't write a regular function that needs quantifers to find succ/pred values. For example, a function *u1-u2-...-un*$\rightarrow$*un...u2u1* where *ux* is a substring
- The main problem is that the BMRS format requires that functions are order-preserving. The YAML file follows this. But I've been thinking of ways to enrich BMRS notation to allow some types of non-order-preserving functions

Open problems

- There's been a lot of new datasets that have been converted to logical transductions and BMRS notation (Strother-Garcia, 2019; Dolatian, 2020; Chandlee and Jardine, 2021)

$\rightarrow$ convert all them to YAML to help find bugs in the YAML format

$\rightarrow$ compile into Python to ensure correctness

Open problems

- There's been a lot of new datasets that have been converted to logical transductions and BMRS notation (Strother-Garcia, 2019; Dolatian, 2020; Chandlee and Jardine, 2021)

→ convert all them to YAML to help find bugs in the YAML format

→ compile into Python to ensure correctness

- BMRS allows operations such as composition and others (Oakden, 2021)

→ create algorithms for doing these operations over YAML formats

→ create algorithms to detect interaction across rules in YAML

Open problems

Open problems

- BRMS and its implementations (YAML format, Python compiler) currently only handle strings...

$\rightarrow$ convert model-theoretic notation for other representations (trees, tiers, gestures, autosegmental graphs) to YAML-friendly syntax

$\rightarrow$ generalize the Python compiler

- BMRS and its implementations (YAML format, Python compiler) currently are just transducers.

$\rightarrow$ would be nice to also handle constraint-checking as an automaton.

Open problems

- BRMS and its implementations (YAML format, Python compiler) currently only handle strings...

$\rightarrow$ convert model-theoretic notation for other representations (trees, tiers, gestures, autosegmental graphs) to YAML-friendly syntax

$\rightarrow$ generalize the Python compiler

- BMRS and its implementations (YAML format, Python compiler) currently are just transducers.

$\rightarrow$ would be nice to also handle constraint-checking as an automaton.

- Spoiler: I'll probably never do these (by myself), but feel free to pitch an idea for collaboration :)

Table of Contents

Conclusion

- It's good practice to double-check your analysis via computational implementations and simulations
- Can now do that with logical transductions
- Implementations open doors for yourself to contribute in this field, whether as a linguist or computer scientist
- Honestly, don't be shy if you're interested in one of the open problems :)

REFERENCES

Bhaskar, S., J. Chandlee, A. Jardine, and C. Oakden (2020). Boolean monadic recursive schemes as a logical characterization of the subsequential functions. In A. Leporati, C. Martín-Vide, D. Shapira, and C. Zandron (Eds.), Proceedings of the 14th International Conference on Language and Automata Theory and Applications (LATA 2020).

Chandlee, J. and A. Jardine (2021). Computational universals in linguistic theory: Using recursive programs for phonological analysis. Language 97(3), 485–519.

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

Oakden, C. D. (2021). Modeling phonological interactions using recursive schemes. Ph. D. thesis, Rutgers The State University of New Jersey, School of Graduate Studies.

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