ProLinguist: Program Synthesis for Linguistics and NLP

Partho Sarthi , Monojit Choudhury* , Arun Iyer* , Suresh Parthasarathy* , Arjun Radhakrishna , Sriram Rajamani

Microsoft Research India

{t-pasar,monojitc, ariy, supartha, arradha, sriram}@microsoft.com

Abstract

We introduce ProLinguist, an approach that uses *program synthesis* to automatically synthesize explicit string transformation rules from input-output examples for NLP tasks. Our algorithm is able to learn rules not only where the output depends on the surrounding input context, but also stateful rules, where it also depends on the results of applying transformation rules to the input context. Our algorithms work for both small and large amounts of potentially noisy training data. Furthermore, the learning process, as well as the level of abstraction of the inferred rules, can be controlled by an expert by providing linguistic knowledge to ProLinguist in the form of a Domain Specific Language. We demonstrate ProLinguist on a variety of NLP tasks ranging from textbook phonology problems to a more complex grapheme-to-phoneme conversion for Hindi and Tamil, showing that it can produce interpretable rules from small amounts of training data.

1 Introduction

String transformations are at the heart of many NLP tasks such as grapheme-to-phoneme (G2P) conversion [Novak *et al.*, 2012], morphological analyzers [Karttunen and Beesley, 2005], transliteration [Knight and Graehl, 1998] and machine translation [Knight, 2007]. Typically, these transformation rules are either hand-coded (e.g., [Choudhury, 2003]), or learned from data in the form of Deterministic Finite-state Automata (DFA) [Beesley and Karttunen, 2003; Casacuberta and Vidal, 2004], decision trees (e.g., [Lee and Oh, 1999]), etc. With the rise of neural networks, sequence-to-sequence models (e.g., RNNs and biLSTMs) are also commonly used for these tasks [Bahdanau *et al.*, 2014]. Here, the transformation rules are learned from large amounts of training data, and are implicitly represented in the structure of the network. However, the two approaches —(1) learning interpretable rules from small amounts of data that would inform a linguist, and (2) learning models from large amounts of training data for developing language application— have largely remained independent and are presently drifting further apart due to complete non-interpretability of neural models.

We use *program synthesis* techniques (see, for example, [Gulwani, 2011; Solar-Lezama *et al.*, 2005; Alur *et al.*, 2013]) to automatically learn string transformation rules from data. The generated rules are interpretable, and the level of abstraction can be controlled by the user by providing an appropriate Domain Specific Language (DSL), which specifies the set of transformation operations. Existing synthesis algorithms can be used to learn rules that transform an input token to a corresponding output token. It is also possible to make the transformation rule depend on the input context, by providing the surrounding input tokens also as additional inputs to the synthesis algorithm. However, learning stateful rules require more than the input context. They require, in addition, the results of the transformations applied to the input context. This is the key insight behind Stateful Noisy Disjunctive Synthesis (Stateful-NDSyn), which is the main algorithm in the paper. Stateful-NDSyn generates stateful rules using multiple passes, and multiple stages in each pass over the input string. In each stage, the results of the outputs of the previous stages are passed in as inputs, and the system learns a function over the history of the past results as well as current input to produce the output. In that sense, our approach transports intuitions from recurrent neural networks (such as RNNs or LSTMs) to program synthesis.

Stateful-NDSyn builds on significant recent advances in program synthesis. FlashMeta [Polozov and Gulwani, 2015] is a powerful framework where the user can input a DSL and the synthesis algorithm restricts the space of rules to the rules expressible by the DSL. By tuning the DSL, the user can convey domain specific intuitions to the synthesis algorithm and learn domain specific rules taking into account such intuitions. Noisy Disjunctive Synthesis (NDSyn), was recently developed to handle noisy labeled data in program synthesis [Iyer *et al.*, 2019]. Our algorithm Stateful-NDSyn, developed in this paper, builds on NDSyn, and learns stateful rules by applying multiple passes as mentioned before. Additionally, since we build on NDSyn and FlashMeta, we are able to handle noise in the inputs, and also allow domain experts to specify domain and language specific intuitions. Previous works [Barke *et al.*, 2019; Ellis *et al.*, 2015] on applying program synthesis to the phonological rule problem are able to

*Contact Authors

learn contextual rules from small noise-free datasets, but not stateful ones.

These new capabilities enable productive use of program synthesis for NLP problems, which has not been possible before. We demonstrate our approach on various lexical and textbook problems as well on a more complex G2P conversion task for two languages —Hindi and Tamil. ProLinguist can learn very accurate and linguistically interpretable rules from an order of magnitude fewer labelled input-output examples as compared to state-of-the-art machine learning systems [Linzen, 2020].

The primary contributions of this work are: (1) We propose a novel stateful program synthesis algorithm called Stateful-NDSyn, to learn linguistic rules. Stateful-NDSyn can be used to learn stateful transformation rules from both small as well as large amounts of data. (2) We demonstrate the applicability of this algorithm for various phonological tasks. We also incorporate an elegant way to include domain knowledge during synthesis. (3) We have implemented the Stateful-NDSyn algorithm in a tool ProLinguist, and demonstrate various aspects of ProLinguist such as generalizability, flexibility, interpretability of the rules, and the ability to handle noise and identify outliers. (Sec 4).

2 Problem Setting

In order to demonstrate the effectiveness of program synthesis in NLP problems, we chose a set of phonological processes across different languages and a well-studied task of G2P conversion. Traditionally, these tasks have been approached through rule-based techniques and therefore, the rules are well documented for many languages. Consequently, unlike more complex sequence-to-sequence tasks, e.g., MT, the chosen tasks provide us a direct way of evaluating and comparing the string transformation programs synthesized by our system against the rules designed by experts.

While program synthesis can be applied to any string transformation problem, phonological processes and tasks like grapheme to phoneme transformation are particularly suited for program synthesis. The transformations are typically independent of other linguistic layers: therefore, the information for learning the re-write/transform rules is present exclusively in the input-output examples. Further, these processes operate on *natural classes* that are universal to most languages (e.g., place of articulation or manner of articulation classes). This allows a program synthesizer to operate on a single domain-specific language, with very few custom per-language features.

Primer on Program Synthesis. Example-based program synthesis is a technique for synthesizing programs from a given DSL that are consistent with a small number of given examples. The key difference between program synthesis engines and other rule induction techniques is that program synthesis engines are able to learn complex programs (rules) from just 1-2 training examples, at the cost of being very domain specific (see, for example, [Gulwani, 2011; Rolim *et al.*, 2017]). In our work, we use the FlashMeta program synthesis algorithm [Polozov and Gulwani, 2015], as implemented in the PROSE framework [Microsoft, 2015]. In the FlashMeta framework, a synthesis task is given by a DSL L and a set of input-output examples $i_k \mapsto o_k$, and the framework synthesizes a program $P \in L$ such that $P(i_k) = o_k$ for each k. As a simple example, L for learning sub-string transformations can look like

```
out := Substring(x, pos, pos);
pos :=  const_int | regex_search(x, r)
```

Given an input-output instance like {`Mr Foo` $\mapsto$ `Mr`}, the synthesizer can output a program like `Substring(x, 0, regex_search(x, '\s'))`.

We refer the reader to the rich literature in this area [Solar-Lezama *et al.*, 2005; Alur *et al.*, 2013; Gulwani, 2011; Polozov and Gulwani, 2015; Singh and Gulwani, 2012], and to [Gulwani *et al.*, 2017] for a survey of techniques. In summary, program synthesis allows us to generate candidate rules from a small number of examples.

Notations. The task at hand is to transform a sequence of input tokens $i_0 i_1 \dots i_n$ to a sequence of output tokens $o_0 o_1 \dots o_m$. We assume that the i_k's and o_k's are drawn from a universe of input and output tokens $\mathcal{I}$ and $\mathcal{O}$, respectively. In the lexical tasks, the universe $\mathcal{I}$ is the set of underlying form, and the set $\mathcal{O}$ is the set of surface form. In the G2P scenarios, $\mathcal{I}$ is the set of orthographic symbols or graphemes in the script of the language, and $\mathcal{O}$ is the set of phones. The program synthesis technique we use produces rules of the general form $A \rightarrow B/X\_Y$

- This notation can be summarized as phoneme or feature vector A is re-written as phoneme or feature vector B when the left context is X and right context as Y [Chomsky and Halle, 1968]. In ProLinguist, the context can be more than a single character.
- To encompass all the rules inferred by ProLinguist, we overload this notation. Specifically, we let A be *graphemes* and X, Y be predicates over graphemes as well as phonemes.
- Also, in cases where A takes *grapheme* values from an Indian language, we denote both the grapheme in native alphabet followed by its ITRANS [Chopde, 2001] notation denoted between ⟨ ⟩ (for example, च⟨*cha*⟩).

3 Program Synthesis for NLP

As stated before, the task of the program synthesis engine is to take as input, examples of the form $i_0 \dots i_n \mapsto o_0 \dots o_m$, and produce rules of the form $A \rightarrow B/X\_Y$. We assume that the alignment between input and output characters is given to us during training.

Given the alignment, we use the term *token-level examples* to denote the input-output behavior of single tokens in the context of a whole word. For example, in the input-output example बचपन ⟨bachpan⟩ $\mapsto$ [bə.ʧ.pə.n], one token-level example is given by च⟨cha⟩ $\rightarrow$ [ʧ] /ब_पन .

Providing Domain Knowledge. We allow a domain expert to provide *domain-specific* features. In Hindi and Tamil, the featural properties used are place of articulation features (e.g., [guttaral], [palatal], and [retroflex]) and manner of articulation features (e.g., [plosive], [nasal] and [fricative]), respectively. These are well-known and universal phonological features that are expected to influence the G2P rules of many languages.

In addition to these, we provide features such as [C], [half vowel], and [full vowel] for whether a token is a consonant, an inherent vowel, or an overtly marked vowel respectively. The last two features are specific to the orthographies of Tamil and Hindi, or more generally - *abugidas*[1]. We also include certain universal phonological features such as [+voi], [−son], [−syll].

Domain specification language Our DSL is equivalent to SPE rules. The main kinds of operators are as follows:

- *Positioning.* We use $\mathsf{relPos}(\mathsf{token}, \mathsf{i})$ operators to identify the context relative to the token of interest. For example, $\mathsf{relPos}(\mathsf{token}, 2)$ and $\mathsf{relPos}(\mathsf{token}, -1)$ refer to the position follows token 2 places to the right, and immediately left of token, respectively.
- *Predicates.* The predicates are Boolean functions on tokens. We use two atomic predicates, and their Boolean combinations. (1) $\mathsf{HasFeature}(\mathsf{F}, \mathsf{token})$ which returns $true$ if token has the feature F, (2) $\mathsf{Match}(\mathsf{y}, \mathsf{token})$ which returns $true$ if token is equal to y.
- *Transformations.* Transformations are given partial functions that map tokens to tokens. We use a set of atomic transformation as detailed below, as well as conditional transformations of the form `if(pred) trans` where pred is a predicate and trans is a transformation.

We list the atomic transformations used in our DSL here: (1) $\mathsf{DefaultTransformation}(\mathsf{token})$ operator provides the default transformation of token including the implicit vowel if any. For example, DefaultTransformation(च) : च⟨cha⟩ $\mapsto$ /ʧə/. (2) $\mathsf{ReplaceBy}(\mathsf{token}, \mathsf{y})$ operator is a general purpose operator which transforms token to y. As an example, in English Past Tense, one of the transformations is ReplaceBy(b,bb) : ⟨b⟩ $\mapsto$ ⟨bb⟩. (3) (Hindi and Tamil) $\mathsf{DeleteSchwa}(\mathsf{token})$ operator deletes the implicit schwa from the default translation of token. We represent this operator as DeleteSchwa(च) : च⟨cha⟩ $\mapsto$ /ʧ/. This transformation can also be represented by using only phonemes as /ə/ $\mapsto$ $\emptyset$. (4) $\mathsf{RetainSchwa}(\mathsf{token})$ operator adds a feature to a token marking that the implicit schwa in the abugida cannot be deleted by additional transformations.

We share the same DSL across all the tasks in our experiments (Sec 4).

3.1 Stateful Noisy Disjunctive Synthesis

NLP problems brings a unique combination of challenges to program synthesis.

- *Noise.* For many languages, the available data-sets themselves are noisy due to the lack of high-quality phonemic transcriptions. In such cases, the dataset itself is built using approximate techniques.
- *Exceptions.* Additionally, languages almost always have exceptions to their standard phonological rules. These may be due to loan words or other in-language exceptions.

[1]An abugida or alphasyllabary, is a writing system in which consonant–vowel sequences are written as a unit

- *Statefulness.* Linguistic rules often interact in a stateful manner. For example, applying one rule to a grapheme will often change what rule should apply on neighboring graphemes.

Example 3.1. In the Hindi G2P case, a well-known stateful rule (see [Choudhury, 2003]) is equivalent to "If the preceding and following characters' schwa is marked as to be retained by previous rule applications, delete the current schwa". Note that this rule is not the same as /ə/ $\rightarrow \emptyset$/ə_ə .

We develop a novel program synthesis algorithm Stateful-NDSyn to synthesize stateful rules from a given set of noisy phonemic transcription examples D.

Algorithm 1 describes the high level procedure for Stateful-NDSyn consisting of three steps in a loop:

- Repeatedly sample a small number (usually 1-3) of examples D' from D, and use a classical (noise-free) program synthesis to produce a large number of candidate rules R'.
- Choose a subset of rules $R \subseteq R'$ that cover all (or most) of the examples in the full data set D with the least amount of errors. This is done with a approximate set cover algorithm.
- Apply the selected rules to the data-set, annotate each token with an additional feature that corresponds to the which rule from R (if any) was applied on that specific token. In the next iteration of the loop, NDSyn may use these additional features to generate rules.

The first two steps are a variant of the NDSyn algorithm from and has been shown to filter out random noise from a dataset [Iyer *et al.*, 2019]. The third step is crucial to synthesizing stateful rules: The annotations allow the following iterations to use the information generated from the current set of rules. We repeat this loop till significant inputs are covered. The output of this algorithm are a stratified set of rules $\mathcal{R} = \langle R_0, R_1, \ldots, R_n \rangle$ along with outliers (Sec 4.3). Intuitively, these rules are applied in sequence, i.e., first apply all rules in R_0, and then R_1 and so on.

Synthesizing Rules from a DSL. The procedure Synth in Algorithm 1 is repeatedly called with a small number of token-level examples D', and produces a candidate rule from the DSL using the FlashMeta synthesis procedure. In all passes but the first, the synthesizer may generate rules that depend on the new annotated features and outputs from the previous passes. An additional complication in this step is the size of the context to allow: choosing very large contexts allows for very specific rules which may only apply in a few words, while very small contexts may not be sufficient to express the required rules. In practice, we start with a context size of 1 (i.e., the tokens preceding and succeeding the current one), and increase the context size over the course of the algorithm.

Selecting rules using Approximate Set Cover. The procedure NDSyn of Algorithm 1 uses the approximate set cover algorithm for picking a few "high quality" rules among the hundreds of candidates generated. The quality of the rule is defined by the Score function based on the number of examples (not already covered by a previously generated rule) that are consistent and inconsistent with the rule. Here, consistent and inconsistent examples are those where the rule applies and

Algorithm 1 Stateful Noisy Disjunctive Synthesis

Require: Token-level examples D
Require: Pass threshold P
$D_{\text{pass}} \leftarrow D, \text{pass} \leftarrow 0$
$\mathcal{R} \leftarrow \langle\rangle$ ▷ Output rule-set sequence
while pass $< P$ **do**
  $R_{\text{pass}} \leftarrow \text{NDSyn}(D_{\text{pass}})$
  $D_{\text{pass}} \leftarrow \forall x \in D_{\text{pass}} \mid \text{Annotate}(R, x)$
  $\mathcal{R} \leftarrow \mathcal{R}, R_{\text{pass}}$ ▷ Add R_{pass} to output sequence $\mathcal{R}$
  pass $\leftarrow$ pass $+ 1$
return $(\mathcal{R}, \text{Outliers(uncovered inputs)})$

function NDSyn(D, Threshold $0 \leq t \leq 1$)
  $R' \leftarrow \emptyset$
  while significant fraction of inputs not covered **do**
    $D' \leftarrow$ Sample small subset of D
    $R' \leftarrow R' \cup \text{Synth}(D', \text{ctx})$
  return ApproxSetCover(D, R', t)

function ApproxSetCover(D, R', t)
  uncovered $\leftarrow D; R \leftarrow \emptyset$
  while |uncovered| $> t \cdot |D|$ **do**
    $r^* \leftarrow \text{argmax}_{r \in R'} \text{Score}(r, \text{uncovered})$
    $R \leftarrow R \cup \{r^*\}$
    uncovered $\leftarrow$ uncovered $\setminus$ {
      ex $\in$ uncovered | Consistent(ex, r^*)}
  return R

function Score(r, S)
  incorrect $\leftarrow \{ex \in S \mid \text{Inconsistent}(ex, r)\}$
  correct $\leftarrow \{ex \in S \mid \text{Consistent}(ex, r)\}$
  if |incorrect| $> \epsilon \cdot |S|$ **then return** 0
  return |correct|

produces the expected and unexpected output, respectively. Note that there is a third category of examples – ones where the rule does not apply at all. Hence, the procedure prioritizes rules that are consistent with a large fraction of the data-set, while making few mistakes.

Example 3.2. Using the case बचपन ⟨bachpan⟩ ↦ [bə.ʧ.pə.n], for the transformation न ⟨na⟩ ↦ /n/, a candidate rule generated is ə → ∅/_# (delete the schwa at the end of the word as the default Hindi transformation is न ⟨na⟩ ↦ /nə/). This rule applies consistently across all examples, and hence, is scored highly and selected early.

Similarly, from च ⟨cha⟩ ↦ /ʧ/, the rule ə → ∅/_C can be generated. However, it is not a standard Hindi phonological rule and is ranked very low due to a large number of inconsistent examples.

Multi-pass to synthesize stateful rules. After each iteration of Algorithm 1, we process the training data with the rules R that were selected, annotating each token with the rule that was used on it (if any). For example, if a token was processed using the DeleteSchwa transformation, we add a feature called DeleteSchwa to the token. In the next iteration, one of the allowed predicates would be HasFeature(DeleteSchwa, _). At each pass i, we attempt to produce the largest set of rules R_{pass} that can be applied without incorrectly transforming more than a fraction ϵ of the tokens. Eventually, after the pass bound P is reached, a sequence of rule sets $R_0, R_1, \ldots R_P$ is returned, along with a number of outliers. The outliers are the set of inputs on which applying all of the returned rules $R_0, \ldots, R_P$ still does not produce the expected output.

Example 3.3. For the Hindi G2P scenario, the synthesizer produces the stateful rule
/ə/ → ∅/RetainSchwa_RetainSchwa
This rule is equivalent to the hand-crafted stateful schwa deletion rule from [Choudhury and Basu, 2002]. For बचपन ⟨bachpan⟩ after the first pass, the output is [bə.ʧə.pə.n]: the characters ब ⟨ba⟩ and प ⟨pa⟩ were processed using RetainSchwa (schwa was retained) and the last character न ⟨na⟩ had its schwa deleted. In the second pass, the above rule deletes the schwa after च ⟨cha⟩: thus producing the final correct output [bə.ʧ.pə.n].

4 Experiments and Results

We evaluated ProLinguist with respect to a number of different criteria:

(a) How does ProLinguist perform on data-sets where transformations are for a single morphological/phonological process? (Section 4.1)
(b) How does ProLinguist perform on noisy grapheme-to-phoneme data-sets that include multiple processes? (Section 4.2)
(c) Are the rules produced by ProLinguist linguistically interpretable, and does it lend itself to easy debugging? (Section 4.3)

4.1 Textbook and Lexical problems

We evaluated ProLinguist on a variety of textbook and lexical problems [Odden, 2005; Gussenhoven and Jacobs, 2017; Farmer and Demers, 2010] (these are the same datasets used for the evaluation in SyPhon [Barke *et al.*, 2019]). The inputs in these tasks are pairs of words in their underlying and surface forms, and the goal is to learn the phonological process for the transformations. The ground truth rules for each of these tasks was taken from [Barke *et al.*, 2019]. Table 1 summarizes the ProLinguist output for a subset of tasks. In the textbook problems, ProLinguist was able to match the accuracy of SyPhon (both 100%). On the other hand, while for the flapping problems, ProLinguist is able to converge to 100% accuracy only with fewer examples as compared to SyPhon — this is due to ProLinguist narrowing down on the context [+stress]_[+syll] with just 20 examples. ProLinguist weights the specificity of contexts significantly higher than SyPhon. In terms of performance, ProLinguist takes between 30 and 170 seconds to learn the rules for the above data-sets as compared to the 5–30 seconds for SyPhon and 1-2 hours for [Ellis *et al.*, 2015]. The slow down as compared to SyPhon can be mostly attributed to ProLinguist using FlashMeta for data-driven deduction-style synthesis as opposed to SMT solving, which is more efficient. However, FlashMeta allows for more expressive and easily customizable DSLs.

4.2 Grapheme to Phoneme Tasks

Our second set of experiments pertain to grapheme-to-phoneme transformation (G2P). All G2P experiments were

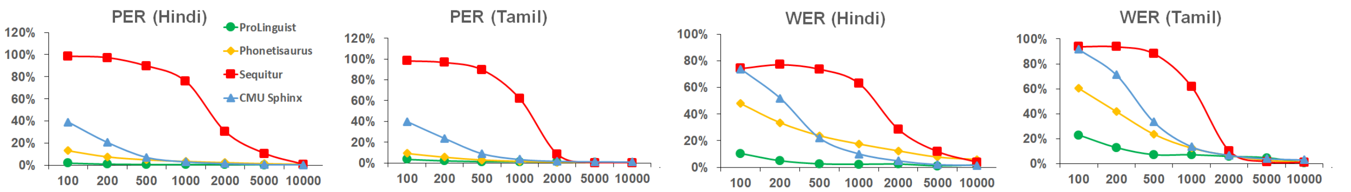

Figure 1: WER, PER for Hindi and Tamil data sets with varying training data sizes.
Legend: ProLinguist Phonetisaurus Sequitur CMU Sphinx

	Data Set	Rules learnt by ProLinguist	
1	English flapping	$\begin{bmatrix}+\text{ant}\\ -\text{voi}\\ -\text{del. rel}\end{bmatrix} \rightarrow [+\text{voi}]\ /\ [+\text{stress}]\ \_\ [+\text{syll}]$	
2	Russian	$[-\text{son}] \rightarrow [-\text{voi}]\ /\_\#$	
3	Scottish	$[+\text{syll}] \rightarrow [+\text{long}]\ /\_ \begin{bmatrix}+\text{cons}\\ +\text{voi}\\ +\text{cont}\end{bmatrix}$	
4	Korean	$\begin{bmatrix}-\text{cont}\\ -\text{voi}\end{bmatrix} \rightarrow \begin{bmatrix}-\text{c.g.}\\ -\text{s.g.}\end{bmatrix}\ /\_\ [+\text{c.g.}]$	
5	Hungarian	$[-\text{son}] \rightarrow [\alpha\text{voi}]\ /\_\ [\alpha\text{voi}]$	
6	Kishambaa	$[+\text{nas}] \rightarrow [-\text{voi}]\ /\_\ [+\text{s.g.}]$	
7	English Past	$\emptyset \rightarrow \langle e\rangle/C\_\#$	$C \rightarrow CC/V\_$
8	English Cont.	$\langle e\rangle \rightarrow \emptyset/\_\#$	$\langle i\rangle \rightarrow \langle y\rangle/C\_$
9	Tohono O'odham	⟨s⟩ → ⟨ṣ⟩/_V	⟨r⟩ → ⟨ḍ⟩/_V

Table 1: Rules learnt for lexical and textbook problems

Language	Rules learnt by ProLinguist	
Hindi Schwa	/ə/ → ə/_CC	/ə/ → ə/#C_
	/ə/ → ə/_[full vowels]	/ə/ → ∅/_#
	/ə/ → ∅/(!DeleteSchwa)C_CV	
	/ə/ → ∅/RetainSchwa_RetainSchwa	
Hindi Anuswara	⟨.n⟩ → ŋ/_[velar]	⟨.n⟩ → ɲ/_[palatal]
	⟨.n⟩ → ɳ/_[retroflex]	⟨.n⟩ → n/_[dental]
	⟨.n⟩ → m/_[labial]	
Tamil Schwa	/ə/ → ∅/_V	/ə/ → ∅/_⟨.n⟩
Tamil Voicing	[−voi] → [+voi] /V_	[+voi] → [−voi] /#_
	[−voi] → [+voi] / [+nasal] _	

Table 2: Inferred G2P rules for Hindi and Tamil

performed on Hindi and Tamil datasets, each containing 25000 words and their corresponding phonetic transcriptions in IPA. The datasets were automatically generated by running G2P converters currently being used in commercial TTS systems on frequently occurring words of the languages. We discovered transcription errors in around 10% of the cases, which were isolated using heuristic rules. Each dataset was then split into two sets: one containing the correct phonetic transcriptions, and the other with erroneous ones. We use the correct set for doing performance and accuracy evaluation and interpretability studies, while using the combined set for debuggability studies (Sec 4.3).

Data. We vary the training data size in the range $\{100, 200, 500, 1000, 2000, 5000, 10000\}$. We create 10 different train-test splits with random seeds 1 to 10 for each data size. The number of tokens varies from 1100 to 120000 for data sizes of 100 and 10000 words respectively. ProLinguist takes $20-40$ seconds for smaller data-sets (100-200), $2-8$ minutes for data-set of size 500, and about 3 hours for the largest data-set (10000). However, as can be seen from Table 1, the accuracy of ProLinguist saturates at 500 examples in the experiments.

Methods. For this task, we compare ProLinguist with three open-source G2P tools: Sequitur [Bisani and Ney, 2008] which is a statistical model, CMU Sphinx [CMU, 2018] which is an LSTM based model and Phonetisaurus [Novak *et al.*, 2012] which is a WFST based model. Note that unlike the other three systems, ProLinguist has access to domain knowledge through the DSL.

Metrics. We use two evaluation metrics: Word Error Rate (WER) and Phoneme Error Rate (PER). Let W be the set of all words in the test set and P be the total number of phones in the transcription of all words in W. For a given word $w \in W$, let $\hat{g}(w)$ and $g^*(w)$ indicate the predicted and gold transcriptions of the word w respectively. The metrics are defined as:

$$\text{WER} = \frac{|\{w \in W \mid \hat{g}(w) \neq g^*(w)\}|}{|W|}$$

$$\text{PER} = \frac{\sum_{w\in W} \text{EditDistance}(\hat{g}(w), g^*(w))}{P}$$

Figure 1 shows the WER and PER numbers for Hindi and Tamil. As we can see, the performance of ProLinguist is comparable to Phonetisaurus and CMU Sphinx when the training data size is large. However, for small training data, ProLinguist outperforms all other methods by a significant margin. For instance, with just 100 training examples in Hindi, ProLinguist achieves a WER of 10.4%, as compared to 47.85% with Phonetisaurus.

Thus, with appropriate domain knowledge, ProLinguist can be guided to learn linguistically sensible and general rules from a handful of examples. This is especially beneficial for endangered and minority languages, where procuring large amounts of labeled examples is difficult, but designing a DSL might be easy due to availability of linguistic documentation for this or other typologically similar languages. This property of ProLinguist makes it a suitable choice for field linguists, who might want to instantly discover rules and exceptions from small amounts of linguistic data, and conduct a more informed data collection.

Inputs covered (%)	20	40	60	80	95	100
# Rules required (Hindi)	2	4	5	8	14	54
# Rules required (Tamil)	1	3	5	9	30	96

Table 3: Coverage of synthesized programs.

4.3 Intrepretability and Debuggability

ProLinguist synthesized on an average 60 and 100 programs for Hindi and Tamil respectively for the input size of 5000. However, as Table 3 shows the top $8-9$ rules cover 80% of the input words. A manual examination of the top $8-9$ rules showed that they correspond to well-known phonological rules in the languages, such as the *anuswara* rules and the schwa retention or obligatory deletion rules as described in Sec 2. The $\langle .\text{n}\rangle \rightarrow$ ŋ/_ [velar] rule applies to around 6% of the examples, which makes it quite a generic rule. ProLinguist also learns the multi-pass schwa deletion rule, as described in Ohala (1983). For Tamil, a top rule that ProLinguist outputs is:

$$[+\text{voi}] \rightarrow [-\text{voi}]\ /\#\_\ [\text{half vowels}]$$

A linguist could interpret this rule as "the consonants at the beginning of the word that are followed by the inherent vowel schwa are rendered unvoiced." In short, ProLinguist is able to discover all the well-documented phonological rules of these two languages with only a $100-200$ examples and a linguistically grounded DSL. We highlight some of the major rules synthesized by ProLinguist in the Appendix.

Debugging. ProLinguist enables debugging by automatically identifying outliers. In addition to the outliers returned by Algorithm 1, words that are processed by rules that apply in very few cases are also of interest for debugging. We examined both these types of words in the Hindi and Tamil G2P dataset, and manually categorized them into three different kinds.

Noisy Data. ProLinguist discovered 2 such cases: (1) *Inconsistent Chandra-bindu handling.* The chandra-bindu diacritic in Hindi forces the previous vowel to be nasalized. In the data, this nasalization was done inconsistently for a small fraction of the cases. (2) *Inconsistent schwa insertion in Tamil.* Similarly, in certain contexts in Tamil an extra (incorrect) schwa was inserted. ProLinguist again learnt a rule to insert this incorrect schwa, but the rule fired in very few instances, marking them as outliers. These were previously unknown limitations of the commercial TTS system we used for generating the train-test data.

Loan words. Words borrowed from other languages often have pronunciations that differ from the norm. For example, in the loan word हैलो ⟨hailo⟩ in Hindi (corresponding to the English word "hello") the ै ⟨ai⟩ is pronounced as /e/ instead of /ɛ/ as in all native Hindi words.

Known Exceptions. Every language has some known exceptions. For instance, in Hindi, schwa is usually pronounced as /ə/. However, in certain rare cases, it is pronounced /e/ (for example, in the word शहर ⟨shahar⟩ ↦ [ʃe.he.r]). These exceptions are also identified by ProLinguist as outliers.

This ability to identify and flag inconsistencies and rarely used rules is an extremely useful facet of ProLinguist. Some of the issues flagged above are extremely hard to discover manually.

4.4 Current limitations

- The running time of ProLinguist increases significantly if a large number of overlapping features are provided. One solution to this problem is to explore the clusters in a ranked fashion during synthesis. We leave this extension to future work.
- ProLinguist outputs rules in the DSL specified in Section 3. However, these programs can be rewritten into SPE notation easily. While this post-processing is currently manual, we intend to automate this in the future. For example, one of the Hindi anusvara rules from Table 2 is produced as `if(hasFeature(relPos(token, 1), `[*palatal*]`)` `replaceBy(token, ⟨.n⟩, /ɲ/)`, and is rewritten as shown in the table.

5 Related Work and Discussion

Machine learning, and particularly deep learning, is the popular approach to most NLP problems these days. For instance, G2P systems have been built using CRFs [Wang and King, 2011; Lehnen *et al.*, 2013], and LSTMs [Rao *et al.*, 2015; Yao and Zweig, 2015; Jyothi and Hasegawa-Johnson, 2017]. Nevertheless, rule-based systems are also central to several NLP tasks such as text normalization, G2P and morphological analysis, where rules already exist or are easy to design by experts.

Deterministic Finite Automata (DFA) are the early rule-learning systems [Casacuberta and Vidal, 2004; Beesley and Karttunen, 2003; Mohri and Sproat, 1996] that can be trained with positive training examples. DFAs have been used in learning rules for G2P [Novak *et al.*, 2012] and morphology [Karttunen and Beesley, 2005]. Statefulness of DFAs provide them sufficient power for representing many linguistic phenomena; however, it is difficult to encode linguistic insights during DFA training.

Decision Trees are a very powerful technique that can learn interpretable *if-then-else* rules. They have been successfully employed in G2P [Andersen *et al.*, 1996; Suontausta and Häkkinen, 2000], text normalization [Raj *et al.*, 2007], prosodic modeling [Lee and Oh, 1999], etc. However, by their very nature, they are state-less classifiers. Modeling statefulness is difficult, though can be done through appropriate feature engineering.

Our method is complementary to the above approaches. A program can be thought of as a DFA, a sequence of if-then-else statements, ranking of constraints, and a combination of any or all of these. In that sense, program synthesis makes little assumptions about the nature of the underlying linguistic phenomena. It is the DSL that provides cues on what are more likely transformations. The higher level of abstraction used by ProLinguist is advantageous on many fronts: amount of data required, interpretability, debugging, and customizability. We believe that our proposed techniques can be used in parallel to machine learning techniques to add a layer of interpretability.

[Brill, 1995] presents an iterative rule-based learning system to minimize the error in local labelling assuming that neighbors are tagged correctly. The process works by enumerating over all transformations and finding the best context to apply it under. The process is Markovian, i.e., the context is given by a single preceding token, and hence, can be enumerated over. In contrast, our work uses a sophisticated program synthesis technique to generate both the transformation as well as the context.

A closely related recent work is SyPhon [Barke *et al.*, 2019] where the authors use constraint-solving based program synthesis to generate rules for phonological processes. This technique is more suited towards noise-free single process tasks with no rule interaction, making the synthesis very efficient. On the other hand, our techniques are mostly focused on handling multiple processes at once using interacting rules. Additionally, SyPhon is restricted to contexts of size 1 while the FlashMeta synthesis framework allows us to handle larger and non-standard contexts.

[Şahin *et al.*, 2020] introduced a dataset where the rules have to be inferred from a very few (typically 5-15) examples. We believe ProLinguist can solve these problems where the DSL contains the appropriate meta-linguistic information which the authors mention as a necessity.

6 Conclusion

We propose a novel program synthesis based technique ProLinguist to generate phonological rules. We have demonstrated the effectiveness of the technique in producing results from small amounts of training data, while providing additional value in the form interpretability and debuggability. ProLinguist can be used to learn interpretable rules even from larger datasets in a scalable way. These results suggest a novel way of combining large uninterpretable models with rule-based systems, by using ProLinguist as an aid in understanding, maintaining, and debugging neural network based models. In the future, we intend to conduct a study into the benefits of using ProLinguist in this manner. Further, we believe that a similar ProLinguist-like system can be used for other NLP tasks such as transliteration and text normalization, and intend to fully explore these possibilities.

References

[Alur *et al.*, 2013] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[Andersen *et al.*, 1996] Ove Andersen, Roland Kuhn, Ariane Lazarides, Paul Dalsgaard, Jürgen Haas, and Elmar Noth. Comparison of two tree-structured approaches for grapheme-to-phoneme conversion. In *Proceeding of Fourth International Conference on Spoken Language Processing. ICSLP'96*, volume 3, pages 1700–1703. IEEE, 1996.

[Bahdanau *et al.*, 2014] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[Barke *et al.*, 2019] Shraddha Barke, Rose Kunkel, Nadia Polikarpova, Eric Meinhardt, Eric Bakovic, and Leon Bergen. Constraint-based learning of phonological processes. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6177–6187, 2019.

[Beesley and Karttunen, 2003] Kenneth R Beesley and Lauri Karttunen. Finite-state morphology: Xerox tools and techniques. *CSLI, Stanford*, 2003.

[Bisani and Ney, 2008] Maximilian Bisani and Hermann Ney. Joint-sequence models for grapheme-to-phoneme conversion. *Speech Communication*, 50(5):434–451, 2008.

[Brill, 1995] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565, 1995.

[Casacuberta and Vidal, 2004] Francisco Casacuberta and Enrique Vidal. Machine translation with inferred stochastic finite-state transducers. *Computational Linguistics*, 30(2):205–225, 2004.

[Chomsky and Halle, 1968] Noam Chomsky and Morris Halle. *The sound pattern of English*. Studies in language. Harper & Row, 1968.

[Chopde, 2001] Avinash Chopde. Itrans: Indian languages transliteration, 2001.

[Choudhury and Basu, 2002] Monojit Choudhury and Anupam Basu. A rule-based schwa deletion algorithm for Hindi. In *Proc. International Conference On Knowledge-Based Computer Systems*, 2002.

[Choudhury, 2003] Monojit Choudhury. Rule-based grapheme to phoneme mapping for Hindi speech synthesis. In *90th Meeting of the Indian Science Congress Association (ISCA)*, 2003.

[CMU, 2018] CMU. Sequence-to-sequence g2p toolkit. `https://github.com/cmusphinx/g2p-seq2seq`, 2018.

[Ellis *et al.*, 2015] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 973–981, 2015.

[Farmer and Demers, 2010] Ann Kathleen. Farmer and Richard A. Demers. *A Linguistics Workbook: Companion to Linguistics*. The MIT Press, 6 edition, 2010.

[Gulwani *et al.*, 2017] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[Gulwani, 2011] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 317–330. ACM, 2011.

[Gussenhoven and Jacobs, 2017] Carlos Gussenhoven and Haike Jacobs. *Understanding Phonology*. Routledge, 2017.

[Iyer *et al.*, 2019] Arun Iyer, Manohar Jonnalagedda, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram Rajamani. Synthesis and machine learning for heterogeneous extraction. In *Proceedings of PLDI*. ACM, 2019.

[Jyothi and Hasegawa-Johnson, 2017] Preethi Jyothi and Mark Hasegawa-Johnson. Low-resource grapheme-to-phoneme conversion using recurrent neural networks. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*, pages 5030–5034. IEEE, 2017.

[Karttunen and Beesley, 2005] Lauri Karttunen and Kenneth R Beesley. Twenty-five years of finite-state morphology. *Inquiries Into Words, a Festschrift for Kimmo Koskenniemi on his 60th Birthday*, pages 71–83, 2005.

[Knight and Graehl, 1998] Kevin Knight and Jonathan Graehl. Machine transliteration. *Computational linguistics*, 24(4):599–612, 1998.

[Knight, 2007] Kevin Knight. Capturing practical natural language transformations. *Machine Translation*, 21(2):121–133, 2007.

[Lee and Oh, 1999] Sangho Lee and Yung-Hwan Oh. Tree-based modeling of prosodic phrasing and segmental duration for korean tts systems. *Speech Communication*, 28(4):283–300, 1999.

[Lehnen *et al.*, 2013] Patrick Lehnen, Alexandre Allauzen, Thomas Lavergne, François Yvon, Stefan Hahn, and Hermann Ney. Structure learning in hidden conditional random fields for grapheme-to-phoneme conversion. In Frédéric Bimbot, Christophe Cerisara, Cécile Fougeron, Guillaume Gravier, Lori Lamel, François Pellegrino, and Pascal Perrier, editors, *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013*, pages 2326–2330. ISCA, 2013.

[Linzen, 2020] Tal Linzen. How can we accelerate progress towards human-like linguistic generalization? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5210–5217, Online, July 2020. Association for Computational Linguistics.

[Microsoft, 2015] Microsoft. Microsoft program synthesis using examples sdk, 2015.

[Mohri and Sproat, 1996] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 231–238. Association for Computational Linguistics, 1996.

[Novak *et al.*, 2012] Josef R. Novak, Nobuaki Minematsu, and Keikichi Hirose. Wfst-based grapheme-to-phoneme conversion: Open source tools for alignment, model-building and decoding. In Iñaki Alegria and Mans Hulden, editors, *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing, FSMNLP 2012, Donostia-San Sebastiían, Spain, July 23-25, 2012*, pages 45–49. The Association for Computer Linguistics, 2012.

[Odden, 2005] David Odden. *Introducing Phonology*. Cambridge Introductions to Language and Linguistics. Cambridge University Press, 2005.

[Ohala, 1983] Manjari Ohala. *Aspects of Hindi phonology*. Motilal Banarsidass, Delhi, 1983.

[Polozov and Gulwani, 2015] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126. ACM, 2015.

[Raj *et al.*, 2007] Anand Arokia Raj, Tanuja Sarkar, Sathish Chandra Pammi, Santhosh Yuvaraj, Mohit Bansal, Kishore Prahallad, and Alan W Black. Text processing for text-to-speech systems in indian languages. In *Ssw*, pages 188–193, 2007.

[Rao *et al.*, 2015] Kanishka Rao, Fuchun Peng, Hasim Sak, and Françoise Beaufays. Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*, pages 4225–4229, 2015.

[Rolim *et al.*, 2017] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 404–415, 2017.

[Şahin *et al.*, 2020] Gözde Gül Şahin, Yova Kementchedjhieva, Phillip Rust, and Iryna Gurevych. PuzzLing Machines: A Challenge on Learning From Small Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1241–1254, Online, July 2020. Association for Computational Linguistics.

[Singh and Gulwani, 2012] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 634–651, 2012.

[Solar-Lezama *et al.*, 2005] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294, 2005.

[Suontausta and Häkkinen, 2000] Janne Suontausta and Juha Häkkinen. Decision tree based text-to-phoneme mapping for speech recognition. In *Sixth International Conference on Spoken Language Processing*, 2000.

[Wang and King, 2011] Dong Wang and Simon King. Letter-to-sound pronunciation prediction using conditional random fields. *IEEE Signal Process. Lett.*, 18(2):122–125, 2011.

[Yao and Zweig, 2015] Kaisheng Yao and Geoffrey Zweig. Sequence-to-sequence neural net models for grapheme-to-phoneme conversion. In *INTERSPEECH 2015, 16th Annual Conference of the International Speech Communication Association, Dresden, Germany, September 6-10, 2015*, pages 3330–3334, 2015.

Appendix

A Synthesized G2P Rules

We mention some of the prominent G2P rules which were synthesized and sorted by frequency of occurrence of the rule. Some of these rules are state-full rules and should be interpreted as

$$\text{A} \rightarrow \text{B}/\text{Pred\_}$$

A is transformed to B if the left context was transformed by a Boolean Pred.

A.1 For Hindi

We provide the rules inferred for Hindi and the tokens covered by them.

Rules	Tokens covered
/ə/ → ∅/_V	19401
/ə/ → ∅/_#	4930
/ə/ → ə/#_C	3265
/ə/ → ə/⟨.h⟩_C	772
⟨.n⟩ → η/_[retroflex]	317
/lə/ → l/[high vowel]_[dental]	31
/hə/ → h/C_	20
/ə/ → ∅/DeleteSchwa_RetainSchwa	20

Table 4: Hindi G2P Rules

A.2 For Tamil

We provide the rules inferred for Tamil and the tokens covered by them.

Rules	Tokens covered
$\begin{bmatrix}+\text{voi}\end{bmatrix} \rightarrow \begin{bmatrix}-\text{voi}\end{bmatrix} /\_\langle .\text{n}\rangle$	15120
$\begin{bmatrix}-\text{voi}\end{bmatrix} \rightarrow \begin{bmatrix}+\text{voi}\end{bmatrix} / \begin{Bmatrix}\text{V}\\ \text{C}\end{Bmatrix} \_ \begin{Bmatrix}\text{V}\\ \text{C}\end{Bmatrix}$	3423
$\begin{bmatrix}+\text{voi}\end{bmatrix} \rightarrow \begin{bmatrix}-\text{voi}\end{bmatrix} /\#\_\text{V}$	2769
$\begin{bmatrix}+\text{voi}\end{bmatrix} \rightarrow \begin{bmatrix}-\text{voi}\end{bmatrix} /\_\#$	278
$\begin{bmatrix}+\text{voi}\end{bmatrix} \rightarrow \begin{bmatrix}-\text{voi}\end{bmatrix}$ /[vallinum]⟨.n⟩_DeleteSchwa	200
⟨ch⟩ → sə/#_DeleteSchwa	82

Table 5: Tamil G2P Rules

B Other Synthesized Rules

The rules mentioned are for English Past Tense and English Continuous. Suffixes ⟨d⟩ and ⟨ing⟩ are dropped. We infer the rules after reducing the transformation to its root form. We mention the rules which require insertion of ⟨e⟩ or duplicate the last character A → AA/X_Y.

Problem	Rules
English Past Tense	∅ → ⟨e⟩/C_# C → CC/V_
English Continuous	⟨e⟩ → ∅/_# ⟨i⟩ → ⟨y⟩/C_
Japanese	/t/ → d/C_

Table 6: Synthesized Rules for Other Problems