Synthesis of Programs from Multimodal Datasets

Shantanu Thakoor[*]
Stanford University
Stanford, CA 94305
thakoor@cs.stanford.edu

Simoni Shah
IIT Bombay
Mumbai, 400076, India
simonisamirshah@gmail.com

Ganesh Ramakrishnan
IIT Bombay
Mumbai, 400076, India
ganesh@cse.iitb.ac.in

Amitabha Sanyal
IIT Bombay
Mumbai, 400076, India
as@cse.iitb.ac.in

Abstract

We describe *MultiSynth*, a framework for synthesizing domain-specific programs from a multimodal dataset of examples. Given a domain-specific language (DSL), a dataset is multimodal if there is no single program in the DSL that generalizes over all the examples. Further, even if the examples in the dataset were generalized in terms of a set of programs, the domains of these programs may not be disjoint, thereby leading to ambiguity in synthesis. *MultiSynth* is a framework that incorporates concepts of synthesizing programs with minimum generality, while addressing the need of accurate prediction. We show how these can be achieved through (i) transformation driven partitioning of the dataset, (ii) least general generalization, for a generalized specification of the input and the output, and (iii) learning to rank, for estimating feature weights in order to map an input to the most appropriate mode in case of ambiguity. We show the effectiveness of our framework in two domains: in the first case, we extend an existing approach for synthesizing programs for XML tree transformations to ambiguous multimodal datasets. In the second case, MultiSynth is used to preorder words for machine translation, by learning permutations of productions in the parse trees of the source side sentences. Our evaluations reflect the effectiveness of our approach.

1 Introduction and Related Work

Programming by Example (PBE) has been a widely studied research area with applications in program synthesis in general, and domain-specific program synthesis in particular. Given a dataset of input-output pairs (or examples), we say that a program P in some program space $\mathcal{P}$ *covers* the example $e = (i,o)$ if P predicts some output for the input i. On the other hand, we say that P *satisfies* the example e, if P correctly predicts its output o on input i. The dataset is *multimodal* with respect to $\mathcal{P}$ if there is no single program in $\mathcal{P}$ that *satisfies* all the examples, else it is *unimodal*. On the other hand, the dataset is *confused* with respect to $\mathcal{P}$ if there does not exist a subset of programs from $\mathcal{P}$ such that each example of the dataset is *satisfied* by every program that *covers* it, and is covered by at least one program. An extreme case is that of an *inherently* confused dataset (*i.e.* confused with respect to any $\mathcal{P}$) where two examples with the same input are mapped to different outputs. For example, consider the machine translation domain, where a sentence may be translated in more than one way. Given a *multimodal* and *confused* dataset, we wish to synthesize a set of programs that *satisfies* each example in the dataset as accurately as possible. Additionally, we would like the set of programs to be as simple as possible (Blumer et al. 1987; Ellis, Solar-Lezama, and Tenenbaum 2015; 2016), in order to minimize the *uncertainty* of its applicability on test data; that is, to ensure that the predictive capability of the program set generalizes well on the test data. Applications for such synthesis include: (i) Semi-automating repetitive data transformations using examples (Gulwani 2011; Raza, Gulwani, and Milic-Frayling 2014; Le and Gulwani 2014; Barowy et al. 2015; Kini and Gulwani 2015) (obtained through logs represented as XML). (ii) The more general problem of Data Wrangling[1] (Gulwani 2016), which is the process of transforming the data from a somewhat unstructured format to a visually appealing and structured format that is more suitable for analysis (iii) Word preordering for machine translation (Bisazza and Federico 2016).

Approaches to PBE vary with the nature of the example dataset. In the case of a dataset and program space, such that there exists at least one program that *satisfies* all examples in the dataset, the synthesis algorithm searches the program space for the (set of) such programs and ranks them based on some criteria. The reader may refer to (Kitzelmann 2011; Gulwani 2016) for some of the recent efforts in this direction. In another case, the dataset may have some outliers (noise), and the synthesis algorithm must disregard such points, after detecting them, while assuming that a single best program *satisfies* all the other (non-noise) examples. A recent work (Raychev et al. 2016), addresses the problem of supervised learning of programs from noisy datasets, by detecting outliers. Their approach involves the use of an iterative feedback mechanism between a program generator and a noise-eliminator. On the other hand, (Ellis, Solar-Lezama, and Tenenbaum 2016) present unsupervised learning of programs from noisy observations based on finding (compressed) data representations that need to be described by programs with small lengths. The idea of biasing program sampling using (small) description lengths has

[*] Work done while at IIT Bombay

[1] Supposedly takes up 80% of the time of data scientists

also been explored for supervised program synthesis (Ellis, Solar-Lezama, and Tenenbaum 2015). In this paper, we address the somewhat different problem of program synthesis from a *multimodal* dataset of examples, that is, a dataset which can be *satisfied* only using two or more programs from the program space. We focus on maximizing accuracy of prediction on the training data by learning to rank the programs for each input in order to minimize errors owing to ambiguity.

(Kitzelmann 2010; 2011) employ Least General Generalization (LGG) (Plotkin 1971a) to learn a functional program for a given incomplete specification, in the form of set of input and output example-pairs. Their system learns the desired program also in terms of user defined functions that may be provided as background knowledge. This system initially treats each input-output example pair as a grounded function and progressively combines all of them into a single generalized function using their LGG. The use of LGGs for programming by examples has found use in domains such as XML transformations (Raza, Gulwani, and Milic-Frayling 2014), string manipulations (Raza, Gulwani, and Milic-Frayling 2015), *etc.* A restrictive assumption about the dataset made therein is that it is possible to find a single program via LGG, that *satisfies* every example in the training set that it *covers*. Effective search of program space and ranking of the *satisfying* programs in order to find the simplest program are the key problems that have been addressed there.

Our Contributions: All of the above mentioned works address the problem of program synthesis either from a *unimodal* dataset or from a *multimodal* dataset in the absence of any uncertainty or confusion. To the best of our knowledge, our domain-agnostic framework for program synthesis (abbreviated as *MultiSynth*) is the first to address programming by example, while accounting for both the confusion and the uncertainty integral to a *multimodal* dataset. We design an efficient and effective algorithm for program set synthesis that provably minimizes confusion measured through program generality. Further, we leverage prior work on learning optimal ranking functions to maximize accuracy in the presence of confusion. We demonstrate the effectiveness of *MultiSynth* in two diverse problem settings, *viz.*, (a) learning XML tree transformations and (b) learning preordering rules for machine translation.

The outline of the rest of the paper is as follows: In Section 2 we provide notations and definitions for the rest of the paper. Section 3 describes the *MultiSynth* framework. The two applications of *MultiSynth* are described in Sections 4 and 5. Section 6 presents the results of our experiments on the two applications. Finally, Section 7 concludes the paper.

2 Preliminaries

We consider the problem of program synthesis in the context of a given domain-specific language (DSL) (Raza, Gulwani, and Milic-Frayling 2014). A DSL provides us with mechanisms to represent elements of the input or output space, and transformations between the two.

In particular, the DSL provides us with syntax for expressing subsets of the input (or output) space. When a subset of the input (or output) space can be represented by a DSL expression, that expression is called a *generalization* of the subset, and elements of the subset are *instances* of the generalization. Each such subset of instances is said to be *subsumed by* the generalization. When it is clear from the context, we shall blur the distinction between a generalization and its instance set and use them interchangeably. An elaboration of a specific DSL for XML tree transformations and the associated subsumption relation can be found in (Raza, Gulwani, and Milic-Frayling 2014).

Let E denote a subset of either the input or output space. A generalization g is the *Least General Generalization (LGG)* (Plotkin 1971b) of a set of subsets $\{E_1,\ldots E_n\}$ of the input (or output) space if (i) each E_i is subsumed by g, and (ii) for any DSL expression g' such that each E_i is subsumed by g', g (or more accurately, its instance set) is also subsumed by g'. For example, in Figure 1, g_1 represents a generalization of slides containing an arbitrary number of textboxes, each with arbitrary text of any size, but of red color. Thus, i_1 and i_2 are subsumed by g_1. In fact, g_1 would be the LGG of i_1 and i_2. It is easy to see that g_2 is also a generalization of i_1 and i_2 (although not their LGG), while g_3 does not generalize them. Note, in passing, the use of variables (e.g. X, Y and Z) in the generalization.

A set E is said to be *generalizable*, if there exists a generalization g in the given DSL which subsumes E. For example, in Figure 1, i_1 and i_2 are generalizable, but i_1 and o_1 are not, since our DSL does not allow for a slide to generalize with a list. Intuitively, generalizing a set of elements may be thought of as giving it a shorter, simpler description than listing all of them.

MultiSynth makes the following assumptions about the generalization mechanism provided by the DSL: (i) generalizability is reflexive, symmetric, and transitive (ie, DSL expressions generalizable together form an equivalence class) (ii) a generalization of a set of DSL expressions (all belonging a particular equivalence class) belongs to the same equivalence class (iii) Two DSL expressions x and y which are generalizable must have an LGG. As we shall see in the following sections, these properties are shared by most naturally encountered DSLs.

A program is a pair of generalizations (g,G) . The *domain* of P is the *instance set* of its input generalization g. We say that P *covers* an input i if g *subsumes* i. The output of a program P for i is obtained by matching g with i, obtaining bindings for variables in g, and instantiating G with those bindings. We also define a notion of a *generality measure* (GM) on programs, where $GM((g,G)) > GM((g',G'))$ when g subsumes g' and G subsumes G'. We can further restrict it to be a submodular [2] function: making a more general program cover a new example should increase GM less than making a less general program cover it. It is clear that the LGG of a set of programs has a higher generality measure than that of any in the set.

For example, if our dataset of example transformations

[2] Recall that a set function $\mathrm{f}(.)$ is said to be submodular if for a given set V and any element $v \in V$ and sets $A \subseteq B \subseteq V \setminus \{v\}$, $\mathrm{f}(A\cup\{v\}) - \mathrm{f}(A) \geq \mathrm{f}(B\cup\{v\}) - \mathrm{f}(B)$. This is called the diminishing returns property.

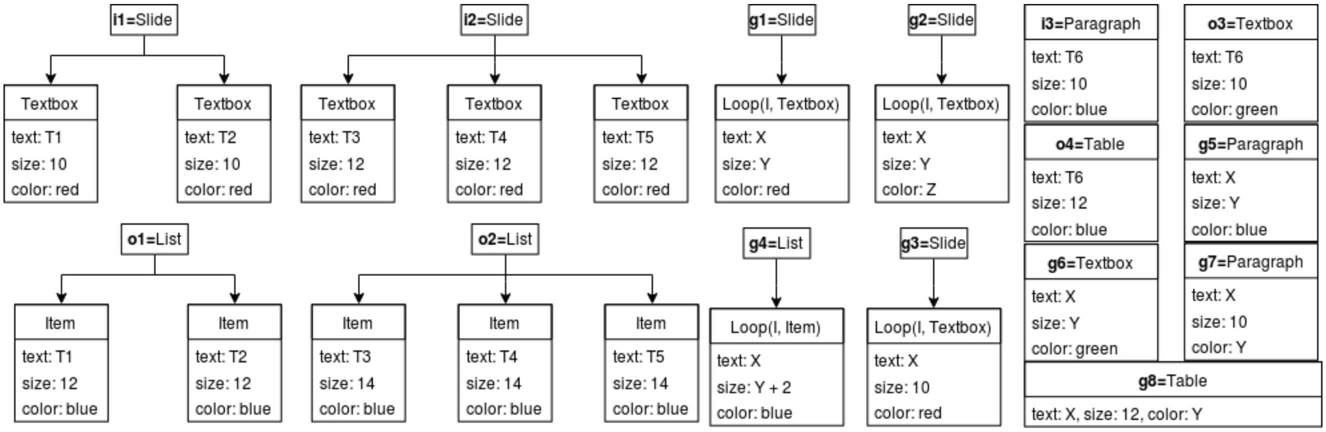

Figure 1: A set of DSL expressions from the XML domain, which will be referred to throughout the paper to provide examples

is $\{(i_1, o_1), (i_2, o_2)\}$ from Figure 1, a program performing those transformation may be (g_1, g_4). Note that X, Y, and I are variables which acquire bindings from the input and are used in the output.

Given a dataset D and a set of programs $\mathcal{S} = \{P_1 \dots P_k\}$, we define $sat(\mathcal{S})$ as the fraction of examples $e = (i, o) \in D$, such that there exists a program $P_j \in \mathcal{S}$ such that $P_j(i) = o$ (that is, P_j *satisfies* e). Clearly, *sat* is a monotone submodular function. We call $\mathcal{S} = \{P_1 \dots P_k\}$ as a *candidate set* for the dataset, if $sat(\mathcal{S}) = 1$.

(i) In general only candidate sets with $k \geq 1$ exist, and we call the dataset *multimodal*. In the special case where one with $k = 1$ exists, we have a *unimodal* dataset.

(ii) We briefly also consider a more general definition that could account for noise level of upper bound $(1 - \theta)$ in D: Given $\theta \in [0, 1]$, $\mathcal{S}$ is a θ-*candidate set* if $sat(\mathcal{S}) \geq \theta$.

It is important to note that because of generalization, there may be cases in which $\mathcal{S}$, while good in other respects, does not partition the input space. Hence, there may be inputs lying in domains of multiple programs that possibly map these inputs to different outputs. Such inputs are said to be *confused* with respect to $\mathcal{S}$. Further, an input is said to be *uncertain* with respect to a set of programs, if it does not belong to the domain of any of the programs. Note that by definition, points which belong to the dataset cannot be uncertain with respect to a candidate set.

Figure 2 gives a pictorial representation of a multimodal scenario, with confusion and uncertainty. Programs P_1 and P_2 have domains that are intersecting. Inputs i_1, i_2, i_3, i_4 lie exclusively in either of the domains, and are deterministically mapped to their corresponding outputs o_1, o_2, o_3, o_4. However, test inputs t_1 and t_2 are *confused*, since they lie in both domains. Their outputs may be either t'_1 or t''_1, and t'_2 or t''_2 respectively. The input s_1 is said to be *uncertain* since the transformation on it is not entirely subsumed by the generalizations of either P_1 or P_2, which take s_1 to s'_1 and s''_1 respectively.

A synthesis algorithm should aim to provide the best possible trade-off between confusion and uncertainty. For example, in the limiting case of generating a separate program for each example data point, accuracy on the training set will be high and confusion will be minimum; however, uncertainty will be very high. This may also be interpreted as a kind of overfitting. On the other hand, when we make the programs more general by generalizing their domains, uncertainty decreases at the cost of increasing confusion. As we shall see later, we use LGGs to limit generalization and ranking to handle any residual ambiguity; thus we strike a balance between confusion and uncertainty.

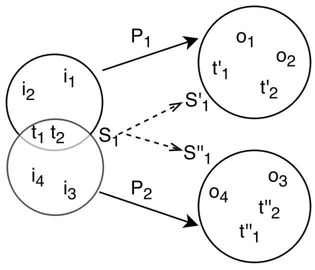

Figure 2: Multimodal Dataset with Confusion and Uncertainty.

In this paper, we consider a framework for dealing with multimodal datasets which are not free from confusion with respect to a (otherwise desirable) candidate set of programs $\mathcal{S}$. Our framework, called *MultiSynth*, describes a general method for synthesizing a set of programs. We use machine learning concepts of generalization, partitioning, and learning feature weights for ranking.

3 The *MultiSynth* Framework

In order to handle the multimodal nature of the dataset, *MultiSynth* associates a program with each mode, and maps each input to the most appropriate mode using a function. More specifically, *MultiSynth* synthesizes a set of programs $\mathcal{S}^* = \{P_1 \dots P_k\} \in 2^{\mathcal{P}}$, and a function f^* which serves as a mapping function. Here $2^{\mathcal{P}}$ is the power set of the program space $\mathcal{P}$. Given an input i, $f^*(i)$ returns the index of the program from $\mathcal{S}^*$ that would best transform i and is implemented as a ranking function over elements of $\mathcal{S}^*$. Hence, the transformation brought about by our model, given an input i, is $P_{f^*(i)}(i)$. Thus, our task reduces to finding a set of programs $\mathcal{S}^*$ and a function f^*.

We aim to strike a balance between confusion and uncertainty. Confusion is central to the multimodal setting, while uncertainty can apply even with unimodality. Most datasets are multimodal, and hence there are many potential transformations for a given point, leading to confusion. On the other hand, uncertainty is the problem of a point corresponding to none of the available modes - not multiple, which is the central problem of multimodal program synthesis. Hence, we motivate a preference in this trade-off, for minimizing confusion rather than uncertainty.

Assuming an upper bound $(1-\theta)$ on the noise level, we restrict $\mathcal{S}^*$ to be a θ-*candidate set*. Further, given our preference for minimizing confusion, we restrict $\mathcal{S}^*$ to have minimal value of the generality measure *GM*. We define the *GM* of a set of programs $\mathcal{S}$ as the sum of $GM(P_l)$ over all $P_l \in \mathcal{S}$.

We model the ranking function f using a weighted linear combination $\mathbf{w}^T\psi(i,P_l)$, where $\psi(i,P_l) \in \Re^m$ is a vector of domain specific features and $\mathbf{w} \in \Re^m$ is a vector of corresponding weights. Note that each feature is a function of the input i and the program being ranked P_l. As for learning the parameters $\mathbf{w}$, one natural choice is to minimize the regularized RankSVM loss (Joachims 2002). Therefore, our optimization formulation can be stated as as

$$\begin{aligned}(\mathcal{S}^*,\mathbf{w}^*) \;=\; &\underset{\mathcal{S},\mathbf{w}}{\text{argmin}} \quad \text{RankSVMLoss}(f_{\mathbf{w}};D,\mathcal{S}) \\ &\text{subject to} \quad \mathcal{S} \in \underset{sat(\mathcal{S}')\geq\theta}{\text{argmin}}\; GM(\mathcal{S}')\end{aligned} \qquad (1)$$

Specifically, the subproblem in the constraint on $\mathcal{S}$ is a case of submodular minimization under cardinality constraints, which is a known NP-Hard problem. A special case is when $\theta = 1$, that is we restrict the search space of $\mathcal{S}$ to *candidate sets*. This amounts to assuming that the dataset is noise-free. For this special case, there exists a unique optimal solution $\{\mathcal{S}^*, f^*\}$ which *MultiSynth* synthesizes in time polynomial in the size of the dataset. Next, we outline the two steps of the algorithm, *viz.*, first identifying the optimal feasible set $\mathcal{S}^*$ and then determining the parameters $\mathbf{w}^*$ of the optimal ranking function f^*_w.

Identifying the optimal feasible set $\mathcal{S}^*$:

(i) *Partitioning:* The training dataset is partitioned, such that for each partition, its set of inputs (and set of outputs) is *generalizable*. This step identifies the modes and the set of points associated with each mode.

(ii) *Generalization and Synthesis:* The i^{th} partition is generalized to program P_i which is expressed as the LGG (as per the DSL) of the inputs in that partition transformed to the LGG of the corresponding outputs.

Given the assumptions on the mechanism for generalization, we note that the partitions created must consist of points all belonging to the same equivalence class (see Section 2). In the special case of $\theta = 1$, we know that every point in D must belong to some partition.

Lemma: Let $\theta = 1$. Partitioning the dataset into equivalence classes based on the *generalizable* relation, leads to our optimal $\mathcal{S}^*$. This can be done in $O(n*p)$ generalizability checks where n is the size of the dataset and p is the number of modes present.

Proof of Lemma: Assume that there exists an $\mathcal{S} \neq \mathcal{S}^*$ such that $GM(\mathcal{S}) \leq GM(\mathcal{S}^*)$. By definition, the set of examples in each partition must be generalizable. Thus, $\mathcal{S}$ must consist of programs synthesized on partitions that were necessarily subsets of the equivalence classes used in $\mathcal{S}^*$. Further, the union of the examples in each partition in $\mathcal{S}$ and $\mathcal{S}^*$ must be equal, as they must both cover each example in the dataset. Hence, the programs in $\mathcal{S}$ must be defined on a partition of the dataset which is a refinement of the partition of the dataset used in $\mathcal{S}^*$. Recall that for any positive submodular function f and disjoint sets X and Y on which it is defined, $f(X) + f(Y) > f(X \cup Y)$. Using this property, we claim that for any program P defined on a set of examples E in $\mathcal{S}^*$, $GM(P) < \Sigma_i GM(P_i)$ where P_i are the programs defined on the subsets of E in $\mathcal{S}$. Hence, it must be that $GM(\mathcal{S}^*) < GM(\mathcal{S})$, which contradicts our earlier assumption. Therefore, $\mathcal{S}^*$ is the unique minimizer of the subproblem in the constraint. Further, it is easy to see that partitioning of the dataset into equivalence classes can be achieved in $O(n*p)$ calls to a domain dependent subroutine that checks if a given pair of expressions is generalizable - we simply start with an empty set of partitions, and one by one either place an example into the unique partition to which it belongs, or create a new partition if it does not generalize with the examples in any existing partition. □

Note that for most applications, we expect p to be much smaller than n, thus letting us find the optimal partition in time almost linear in the size of the dataset. The time required for checking generalizability for a given pair of DSL expressions will be dependent on the DSL.

The process of generalization can give rise to *confusion* during prediction and we illustrate this point through an example. With respect to Figure 1, consider a dataset containing blue paragraphs being changed to green textboxes, and paragraphs of size 10 being changed to tables of size 12. The partitioning step would recognize these as distinct transformations, and create two partitions for the datapoints. After generalization, the programs learned would be (g_5, g_6) and (g_7, g_8). Now, i_3 would be a confused datapoint with respect to the two programs learned. This confusion could be resolved by the ranking function $f_{\mathbf{w}}$ through the design of an appropriate $\psi(i,P_l)$ vector consisting of features such as the number of datapoints in partition P_l, the extent of similarity between i_3 and input points in P_l, and so on. Based on the model $f^*_{\mathbf{w}}$ obtained via learning to rank, we can then decide whether to transform i_3 to o_3 or o_4. Hence, we try to remove confusion through minimizing generality.

Determining the ranking function $f^*_{\mathbf{w}}$:

(i) *Designing & Computing Feature Vector $\psi(i,P_l)$:* The domain-specific features may be properties of the input or the partition alone, or measures of the extent of "matching" between the input and the partition.

(ii) *Learning to Rank:* We learn weights on the above features by leveraging an existing cutting-plane algorithm (implementation) for the convex regularized RankSVM (Joachims 2002) formulation.

Additional Notes: In addition to handling confused datapoints, we can handle uncertain inputs by testing whether the input is "partially subsumed" by the input generalization of some program, and choosing to transform by the "best" of those partially subsuming programs. For example, if we have a program that takes blue text of size X font and converts it to size $X+2$; red text of size 10 would be partially subsumed in the input space, but since the output depends on X and not on the color blue, we can still transform it to red text of size 12. While (Singh and Gulwani 2016) provide the capability to partition the data based on the existence of LGG on the input side, such a strategy does not guarantee the existence of LGG on the corresponding set of outputs. We take care of this by design, through partitioning based on transformations. Further, generalization/partitioning on the input can also lead to confusion as pointed out in Figure 2, something we handle in this work.

4 Program Synthesis for XML Transformations

We now describe the domain-specific parts of our framework used in the XML transformation domain. These include the DSL and generalization mechanisms, and the features for ranking.

DSL & Generalization: We use the DSL in (Raza, Gulwani, and Milic-Frayling 2014) and summarize it here. A rooted tree expression, denoted as $(e,\phi)[\tau]$, consists of a node e (eg. textbox, table, paragraph, etc.) and a list of trees $[\tau]$ that form its children. In addition, the node also contains a map ϕ from the attributes of e to their values. Generalization of trees is achieved by (i) allowing some nodes in the tree to be iterators, which represent an arbitrary number of instances of a particular tree expression (ii) allowing ϕ to map a field name to a don't care value denoted by a variable. For example, in Figure 1, g_3 represents a slide containing an arbitrary number of textboxes, each containing some red colored text of size 10.

A concrete tree τ_c is said to *match* a tree expression τ, if τ_c can be obtained from τ through a substitution that replaces variables by concrete trees and iterators by other substitutions. The matching relation can be generalized to *subsumption* over tree expressions: A tree expression τ subsumes τ' if τ' can be obtained from τ by substituting variables with *tree expressions* (instead of concrete trees) and iterators by substitutions.

Once the dataset is partitioned and the inputs and outputs of each partition are generalized to their respective LGGs, the synthesis algorithm then infers relationships (Raza, Gulwani, and Milic-Frayling 2014) between the variables and iterators of the input and output tree expressions. A program is successfully synthesized if the output tree expression only contains variables and iterators from the input tree expression.

Features: The features used when deciding which program to use for a given input are as follows: (i) number of datapoints in the partition, (ii - v) the number of iterators/variables in the partition LGG before/after generalizing with the input, (vi, vii) the number of literals and variables that matched between the LGG and the input.

5 Program Synthesis for Machine Translation

Approaches to Machine Translation (MT) have been known to benefit from preordering (ie, permuting the words of a source-side sentence before translation), using both learned models (Genzel 2010) as well as hand-written rules (Collins, Koehn, and Kučerová 2005). Rule and memory based MT systems depend heavily on their algorithms for (a) matching chunks from the input sentence with those in their glossary or rules and (b) aligning the translations of these chunks. Many of these approaches, however, treat the input sentences either as plain text or as Part-Of-Speech tag sequences at best.

We explore the problem of preordering as a tree transformation problem, by learning rules for transforming the parse tree of the input sentence. We view the preordering rule extraction problem as a problem of program synthesis from examples, and employ *MultiSynth* for the same. In other words, the problem of learning preordering rules for source language sentences, is perceived as that of learning a deterministic program that preorders the input sentence's parse tree so that it more faithfully represents the ordering of words in the target sentence.

While such a preordering involves a variety of tree transformations, in order to demonstrate the effectiveness of *MultiSynth* in preordering for MT, we restrict the scope of the problem to that of learning transformations at the production level of the parse tree; *i.e.*, permutation of children at every node of the source-side parse tree. Such a program takes a production in a given parse tree as input, and returns the permutation of the RHS of the production as output. Such tree-based transformations of the source-side sentence are known to be effective means of preordering and have been well studied (Genzel 2010; Lerner and Petrov 2013).

We illustrate the extent of multimodality and confusion in this domain, on the CoNLL dataset (Khapra and Ramanathan 2012), which is frequently used to train preordering models. Of the 1131 different possible productions in the English to Farsi component, there are 167, 26, 9, 11 and 5 productions, which respectively have 2, 3, 4, 5, 6 possible ways in which they are permuted, with a large number of training data points in each case.

The training data is preprocessed to acquire the syntax trees of the source and target side sentences, using a standard source side parser (such as (Klein and Manning 2003)), and a bottom-up construction of the target tree, with the source tree as reference. Further, all the productions, their features and their corresponding ideal permutations serve as the training dataset to the synthesizer.

DSL & Generalization: The DSL for syntax tree transformations using production-level permutations includes the following:

(i) Terminal symbols (which include all words in the source side sentence), non-terminal symbols (POS tags) and variables (which may take terminal or non-terminal symbols as values). A production has a non-terminal symbol in the LHS and a sequence of terminal/non-terminal symbols in the RHS, and defines the grammatical rules of a language.

The root of a syntax tree is a non-terminal symbol representing the sentence. For any parent node and its ordered list of children, there must exist a valid production having the parent node as the LHS and the children as the RHS.

(ii) A generalized tree is a syntax tree with either terminal symbols or variables at the leaves.

(iii) The set of primitive operations, which are all the possible permutations of the various productions.

Our generalization mechanism is as follows: given two trees t_1 and t_2, let $G(t_1,t_2)$ denote the generalization of t_1 and t_2. Let $r(c_1 \dots c_n)$ denote a tree rooted at r, whose children, along with their subtrees is denoted by the list $c_1 \dots c_n$. Then, $G(r_1(c_1 \dots c_n), r_2(d_1 \dots d_m))$ is given by the following recursive definition:

(i) $r_1(G(c_1,d_1) \dots G(c_n,d_n))$, if $r_1 = r_2$ and $n = m$

(ii) X, a variable which can take values from $\{r_1, r_2\}$, otherwise

The above definition may be extended to a generalization over a set of trees, as well.

A *node* is a non-terminal or terminal present in a syntax tree. We define as the *content* of a node in a tree, the subtree rooted at it. The *context* of a node is a tuple consisting of its parent, a list of its parent's children, and an integer indicating the index of the node in the list. The *depth* of a node is its distance from the root node of the syntax tree.

Features: The following measures of the extent of matching between a given test input and a candidate partition, are used as features: (f1) subsumption: is a binary value which indicates whether the content (subtree) of the test input production is subsumed by the LGG of the content of points in the candidate partition. (f2) matchscore: a value between 0 and 1, to indicate the extent of subsumption of test input's content in the LGG content of points in the candidate partition. (f3) relative frequency: the size of the candidate partition with respect to the total size of all candidate partitions (f4) number of candidate partitions, for the given input production (f5) relative depth of the input production in the input parse tree (f6) context: binary value, whether the context is in the list of contexts of the candidate production or not (f7) context frequency: the number of training instances in the candidate partition which have the same context as that of the test input.

6 Evaluations

In all our comparisons, our baseline consisted of the results obtained by applying the existing techniques for unimodal program synthesis (Raza, Gulwani, and Milic-Frayling 2014).

Another benchmark we considered was Markov Logic Networks (MLNs), that form a state-of-the-art relational learning framework to simultaneously learn features (formulae in first order logic) and parameters (feature weights) for classification tasks. We posed our problem within the MLN framework by formulating a classification task where each class label corresponds to a transformation. We subsequently explored the space of features and parameters through a popular implementation of MLNs (Kok and Domingos 2009), *viz.*, Alchemy2 and learned them for different choices of search and optimization parameters. We encountered one of two challenges with each execution of Alchemy's beam search for the appropriate structure: (a) for the beam size (threshold) parameter up to a certain value in the thousands, the model with the resultant features yielded accuracies exactly coinciding with the baseline whereas (b) for beam size exceeding the threshold, the process terminated with segmentation fault on all machines with a variety of configurations that we experimented on.

Domain: XML Transformations: We consider the following set of problem statements (P1-P7) to be synthesized. The statement of the problems have been extracted from online help forums for various types of transformations, and repetitive tasks, for which automation would greatly improve the efficiency of the end user. Problems P1, P2 and P3 respectively match the problem statements T11, T15 and T19 that are described in (Raza, Gulwani, and Milic-Frayling 2014). The problem statements are as follows:

P1 - Convert textbox into table, list or textbox depending on the size, font and color of the text inside the textbox.

P2 - Proper alignment of images based on number of images present in a slide.

P3 - Change bullet colors in list depending on the color of the bullet itself and the color of the neighbor bullets.

P4 - Change color, size of text in slide depending on the alignment of the image inside the slide.

P5 - Change text style (bold, underline, italic) in textbox depending on font, size of text.

P6 - Convert list into table, textbox or list depending on bullet type, size and font of text in list.

P7 - Change text indentation in table depending on the border, font of text inside table.

For each of the above problem statements, datasets are simulated, under the following scenarios S1-S5. S1 is the confusion-free case, while S2, S3, and S4 contain 20%, 40%, and 60% respectively of confused data points in both tuning and testing data. S5 contains 40% confused and 20% uncertain data points. The accuracies reported in Table 1 are on an average, 30% above their corresponding baseline accuracies. In particular, in S5, the case with uncertainty, we found an average improvement of 36% over the baseline.

For each problem (P1-P7), training data (used for learning $\mathcal{S}^*$) of size 100 [3] and 3-4 modes, is simulated, with 3-4 attributes, each having 4-5 values. The values for the attributes are chosen randomly from a prescribed set of values. Under each scenario, tuning (used for learning f_w^*) and test data of sizes 300 and 100 examples respectively, are simulated.

Under scenario 1 (free of confusion), *MultiSynth* reported 100% accuracy for all problems P1-P7. For scenarios S2-S5, the accuracy (as percentage) in prediction on tuning (when tuning data is given as test data) as well as test data, are tabulated as in Table 1. The "-" entries for P2, come from the fact that the partitions cover all possible data points, and hence there were no uncertain data points. P1, P2, P4 require on an average 30 seconds, including partitioning on training data,

[3] With rampant rise in the usage of collaborative authoring and editing platforms, dataset sizes in the orders of 100s are becoming common.

	Tun S2	Tun S3	Tun S4	Tun S5	Tst S2	Tst S3	Tst S4	Tst S5
P1	96	93	90	93	96	96	90	95
P2	100	100	100	-	100	100	100	-
P3	80	75	63	70	80	76	53	67
P4	98	88	84	92	99	91	85	89
P5	89	83	71	73	91	82	74	79
P6	89	99	95	97	88	98	93	96
P7	93	83	73	77	89	80	72	75

Table 1: Evaluation of *MultiSynth* for XML Transformation

	Tun	Test	Baseline
Farsi	90.53	89.55	85.17
Italian	91.15	90.35	86.70
Urdu	90.33	90.65	87.33

Table 2: Evaluation of *MultiSynth* for Machine Translation

learning weights on tuning data, and getting the accuracy on tuning and test data for all scenarios S2-S5. P5, P6, P7 require on an average 25 seconds. P3 requires 15 seconds.

Domain: Machine Translation: For each of the pairs (English-Farsi, English-Italian, English-Urdu), we report accuracy over productions which require permutation of the RHS nodes to achieve the ideal reordering. We performed experiments on the settings described in the CoNLL reordering task (Khapra and Ramanathan 2012). The accuracies in reordering over tuning and test data are summarized in Table 2. They are also compared with the baseline to demonstrate the improvement owing to *MultiSynth*'s approach.

The weights learnt for the features f1-f7 in each of the cases, are presented in Table 3 The weights give us insight into the nature of the translation problem in different languages. For example, f5 (relative depth) and f7 (context frequency) having a lower value in Urdu indicates that the way a production is reordered is largely independent of where in the sentence it occurs; on the other hand, Farsi seems to be a language where reordering rules would be highly sensitive to context. Such insights may be used in the design of other translation models for such languages.

In addition to accuracy, we also measured *MultiSynth*'s performance using BLEU scores. (Papineni et al. 2002), a popular metric for evaluating the quality of machine translation output. As mentioned in Section 5, we restrict our problem to sentences which can be correctly translated using tree-based transformations. On these sentences, we obtained BLEU scores of 77.51 for En-Fa, 79.84 for En-It, and 65.35 for En-Ur, with baselines of 14.12, 20.07, 15.85 respectively. In Table 4, we compare BLEU scores against the available results of existing systems (Kunchukuttan and Bhattacharyya 2016; Patel et al. 2016; Dlougach and Galinskaya 2013; Visweswariah et al. 2011) in the shared task from (Khapra and Ramanathan 2012). However, over 80% of the sentences in this dataset are not tree-based transformations, accounting for a drop in performance. Further, our system uses only a source side constituency parser whereas the other pre-reordering techniques additionally require parsing information for the target language. Despite this, our scores compare favorably to specialized machine translation systems. On the other hand, our work could complement recent success (Wu, Zhou, and Zhang 2017) of end to end sequence to sequence neural translation models that have shown to benefit from source side syntactic analysis.

	f1	f2	f3	f4	f5	f6	f7
Fr	.71	13.1	-0.08	-1.88	4.5	1.42	.09
It	.22	12.3	-0.03	-1.01	4.4	-0.65	.08
Ur	.69	7.0	-0.04	-0.03	2.1	-0.19	.00

Table 3: Feature weights learnt for Machine Translation

	Fa	It	Ur
Baseline	50.0	65.1	38.3
Kunchukuttan et al. 2016	46.4	64.7	37.8
Gupta et al., 2016	55.7	73.0	44.7
MultiSynth	62.5	73.8	56.8
Dlougach and Galinskaya, 2013	65.6	76.7	55.8
Vishwesariah et al., 2011	68.7	83.0	63.3

Table 4: BLEU score comparison with other systems

7 Summary and Conclusion

We propose *MultiSynth*, a framework for synthesis of program sets in a DSL, given a multimodal dataset, while balancing confusion and uncertainty. The approach adopted is to reduce confusion by constraining the search space of program sets to those that have least possible generality measure, while satisfying at least $\theta \in [0,1]$ fraction of the training examples. By virtue of properties of the generality measure and the satisfaction function, we were able to develop a 2-stage polynomial time algorithm that found the unique solution for $\theta = 1$: (i) partition the training data based on generalizability of input-output pairs, where each partition corresponds to a mode (ii) use the LGG within each partition to synthesize the corresponding program. In a subsequent step, we resolved the confusion of mapping a data point to an appropriate partition by learning a function that ranks programs from partitions that contend for the data point.

The specific contributions of this paper are (i) the efficient synthesis of generalized programs for multimodal datasets, through LGGs (ii) the effectiveness of learning to rank on confused data points, illustrated on two application domains. For future work, one may consider the problem of multimodal program synthesis in the presence of noise, i.e. when $\theta < 1$. For example, one may try to devise an approximation algorithm for the constrained submodular minimization subproblem in our optimization function, and hence quickly find a good partition of the dataset to synthesize programs.

Acknowledgement: We acknowledge Ajit Diwan, Anamaya Tengse and Animesh Baranawal for contributions.

References

Barowy, D. W.; Gulwani, S.; Hart, T.; and Zorn, B. 2015. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *ACM SIGPLAN Notices*, volume 50, 218–228. ACM.

Bisazza, A., and Federico, M. 2016. A survey of word reordering in statistical machine translation: Computational models and language phenomena. *Computational Linguistics*.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's razor. *Information processing letters* 24(6):377–380.

Collins, M.; Koehn, P.; and Kučerová, I. 2005. Clause restructuring for statistical machine translation. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, 531–540. Stroudsburg, PA, USA: Association for Computational Linguistics.

Dlougach, J., and Galinskaya, I. 2013. Building a reordering system using tree-to-string hierarchical model. *CoRR* abs/1302.3057.

Ellis, K.; Solar-Lezama, A.; and Tenenbaum, J. 2015. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, 973–981.

Ellis, K.; Solar-Lezama, A.; and Tenenbaum, J. 2016. Sampling for bayesian program learning. In *Advances In Neural Information Processing Systems*, 1289–1297.

Genzel, D. 2010. Automatically learning source-side reordering rules for large scale machine translation. In *Proceedings of the 23rd international conference on computational linguistics*, 376–384. Association for Computational Linguistics.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, 317–330. ACM.

Gulwani, S. 2016. Programming by examples (and its applications in data wrangling). In Esparza, J.; Grumberg, O.; and Sickert, S., eds., *Verification and Synthesis of Correct and Secure Systems*. IOS Press.

Joachims, T. 2002. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 133–142. ACM.

Khapra, M. M., and Ramanathan, A. 2012. Report of the shared task on learning reordering from word alignments at rsmt 2012. In *24th International Conference on Computational Linguistics*, 9.

Kini, D., and Gulwani, S. 2015. Flashnormalize: Programming by examples for text normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, 776–783. AAAI Press.

Kitzelmann, E. 2010. *A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs*. Ph.D. Dissertation, Universität Bamberg.

Kitzelmann, E. 2011. A combined analytical and search-based approach for the inductive synthesis of functional programs. *KI-Künstliche Intelligenz* 25(2):179–182.

Klein, D., and Manning, C. D. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, 423–430. Association for Computational Linguistics.

Kok, S., and Domingos, P. 2009. Learning markov logic network structure via hypergraph lifting. In *Proceedings of the 26th annual international conference on machine learning*, 505–512. ACM.

Kunchukuttan, A., and Bhattacharyya, P. 2016. Faster decoding for subword level phrase-based SMT between related languages. *CoRR* abs/1611.00354.

Le, V., and Gulwani, S. 2014. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, 542–553. ACM.

Lerner, U., and Petrov, S. 2013. Source-side classifier preordering for machine translation. In *EMNLP*, 513–523.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, 311–318. Association for Computational Linguistics.

Patel, R. N.; Gupta, R.; Pimpale, P. B.; and M, S. 2016. Reordering rules for english-hindi SMT. *CoRR* abs/1610.07420.

Plotkin, G. D. 1971a. A further note on inductive generalization. *Machine intelligence* 6(101-124).

Plotkin, G. D. 1971b. A further note on inductive generalization. *Machine intelligence* 6:101–124.

Raychev, V.; Bielik, P.; Vechev, M.; and Krause, A. 2016. Learning programs from noisy data. In *ACM SIGPLAN Notices*, volume 51, 761–774. ACM.

Raza, M.; Gulwani, S.; and Milic-Frayling, N. 2014. Programming by example using least general generalizations. In *AAAI*, 283–290.

Raza, M.; Gulwani, S.; and Milic-Frayling, N. 2015. Compositional program synthesis from natural language and examples.

Singh, R., and Gulwani, S. 2016. Transforming spreadsheet data types using examples. In *ACM SIGPLAN Notices*, volume 51, 343–356. ACM.

Visweswariah, K.; Rajkumar, R.; Gandhe, A.; Ramanathan, A.; and Navratil, J. 2011. A word reordering model for improved machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, 486–496. Stroudsburg, PA, USA: Association for Computational Linguistics.

Wu, S.; Zhou, M.; and Zhang, D. 2017. Improved neural machine translation with source syntax. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 4179–4185.