Compositional Program Synthesis from Natural Language and Examples

Mohammad Raza  
Microsoft Research Cambridge  
a-moraza@microsoft.com

Sumit Gulwani  
Microsoft Research Redmond  
sumitg@microsoft.com

Natasa Milic-Frayling  
Microsoft Research Cambridge  
natasamf@microsoft.com

Abstract

Compositionality is a fundamental notion in computation whereby complex abstractions can be constructed from simpler ones, but this property has so far escaped the paradigm of end-user programming from examples or natural language. Existing approaches restrict end users to only give holistic end-to-end specifications, which limits the expressivity and scalability of these approaches to relatively simple programs in very restricted domains. In this paper we propose a new approach to end-user program synthesis where input can be given in a compositional manner through a combination of natural language and examples. We present a domain-agnostic program synthesis algorithm and demonstrate its application to an expressive string manipulation language. We evaluate on a range of complex examples from help forums that are beyond the scope of previous systems.

1 Introduction

End-user programming aims to empower the vast majority of computer users who are non-programmers with the ability to program computers. This may be achieved through natural interaction techniques such as programming by example (PBE), programming by natural language (PBNL) or a combination of such approaches. The challenge is to translate descriptions of tasks in such natural form into a program in an underlying domain specific language (DSL) that is hidden from the user. Although there have recently been successful commercial applications for synthesizing simple programs in application domains such as spreadsheets [Gulwani, 2011] or other office applications [Raza et al., 2014], the main obstacle faced by existing approaches is scalability to sophisticated programs in expressive DSLs.

In PBE approaches [Lieberman, 2001; Gulwani et al., 2012], where the aim is to generate programs from a small number of input-output examples, the performance degrades as the DSL becomes more expressive, since there can be many possible programs satisfying a small set of examples. Hence such approaches usually impose a strong language bias to restrict the domain of possible programs. For instance, the state of the art PBE system of [Gulwani, 2011] which is available in Microsoft Excel 2013, permits only a very restricted subset of regular expressions (without literal string tokens, disjunctions, or iterations), disallowing many common programs including search-and-replace operations. We refer to such limitation on the DSL as the expressivity bottleneck. On the other hand, PBNL approaches [Gulwani and Marron, 2014; Manshadi et al., 2013; Kushman and Barzilay, 2013] have the potential to support more expressive DSLs, as they enjoy a stronger preference bias coming from explicit natural language descriptions of intent as opposed to just examples. However, apart from issues such as ambiguity in natural language specifications, such approaches are limited by the adequacy of the training phase - the supervision bottleneck - a general problem in the field of semantic parsing [Clarke et al., 2010].

With regard to these scalability bottlenecks, a notable characteristic of existing PBE/PBNL approaches is the lack of compositionality in the interaction paradigm. End users can only give a holistic specification of the entire task, be it an NL description or input-output examples describing the whole task at once. In contrast, compositionality is at the heart of programming, with fundamental abstractions such as expressions, procedures, classes or libraries providing modularity and separation of concerns that helps complex programs to be constructed from simpler ones. We also observe the need for such compositionality when end users express their requirements to expert programmers in online help forums: often for sophisticated tasks, the expert must request elaboration or examples about specific parts of the user’s initial task description before they can provide a satisfactory answer.

Hence as tasks get more sophisticated, some kind of compositional paradigm is the way forward. But the reason this is difficult to achieve in end user settings is that the user is not aware of the nature of the underlying formal DSL, and therefore has no guidance on how to break a task down into appropriate subtasks. In this paper we propose to address this issue by leveraging the compositionality that is present in natural language itself. Natural language descriptions of complex tasks commonly refer to constituent concepts in the form of noun phrases that occur in the sentence. Thus using standard techniques to analyse the phrase structure of the natural language descriptions, in our approach we allow the user to provide examples not just of the input and output of the whole program, but of the constituent abstractions as well.
Such intermediate examples are used by the system to synthesize relevant program components to improve the accuracy and performance of synthesis, without sacrificing DSL expressivity (as in PBE) or being limited by training data (as in PBNL).

For instance, consider the task from a help forum shown in Figure 1, where the user needs to match a complex string pattern. This requires a regular expression involving disjunction, concatenation, different forms of iteration and literal character tokens, and cannot be synthesized by existing PBE or PBNL systems such as [Manesh et al., 2013; Gulwani, 2011]. The figure shows the original NL task description from the help forum, and four input-output examples which we gave for the task (where Ex 4 is a negative example). The figure also shows three noun phrases detected by the Stanford phrase structure parser [Klein and Manning, 2003], and the corresponding examples we gave for each of these constituent concepts. With this input, our system synthesizes the program using the desired regular expression.

Since our system uses an expressive DSL supporting unrestricted regular expressions, there could be many possible programs satisfying the 4 examples given. However, using the intermediate examples of the constituent concepts, the system first generates relevant components such as the interval expression Interval(NumChar, 1, 5) for the concept “1-5 numbers”, Interval(NumChar, 4) for “4 numbers” and the single character token UpperChar for “a single letter”. Using these relevant concepts, the system is able to generate the correct program efficiently and rank it higher than other programs that may be using different components.

An important point to note is that in this approach we are not using any natural language learning techniques commonly used in PBNL approaches, but are only utilising the natural language decomposition to enable compositional input in the form of constituent examples. We avoid any supervised language learning in order to evaluate the effectiveness of the compositional approach independently of other approaches and without reliance on any form of training input. However, in practice we envision an ideal system combining the strengths of the two approaches: semantic language understanding may alleviate the need for compositional input in common cases, but for sophisticated tasks where language understanding may fail due to ambiguity or inadequacy of training, the user can still achieve their goal through compositional input.

We begin in the next section by describing the general program synthesis framework and illustrate it with a particular instantiation for the domain of string transformations. The abstract framework consists of three main concepts: a domain-specific language (DSL), the notion of a compositional specification for tasks, and the notion of a component satisfaction relation (CSR) which formally relates DSL components to a compositional specification. In the following section we describe the domain-agnostic program synthesis algorithm which, parametrized by a DSL and a corresponding CSR, generates programs from compositional specifications of tasks. We then present an evaluation of our technique on complex examples from online help forums and end with a discussion of related work and research outlook.

\[ "G" \text{ followed by 1-5 numbers or } "G" \text{ followed by 4 numbers followed by a single letter } "A"-."Z" \]

Examples:

<table>
<thead>
<tr>
<th>Input</th>
<th>Ex 1</th>
<th>Ex 2</th>
<th>Ex 3</th>
<th>Ex 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1-5 numbers”</td>
<td>G2</td>
<td>G12345</td>
<td>G1234B</td>
<td>G123456</td>
</tr>
<tr>
<td>“4 numbers”</td>
<td>G2</td>
<td>G12345</td>
<td>G1234B</td>
<td>null</td>
</tr>
<tr>
<td>“a single letter”</td>
<td>G2</td>
<td>12345</td>
<td>1234</td>
<td>B</td>
</tr>
</tbody>
</table>

Synthesized program:

\[
\text{Filter(}
\text{DisjTok(}
\text{ConcatTok(}
\text{CharTok("G"),}
\text{Interval(NumChar, 1, 5)),}
\text{ConcatTok(}
\text{CharTok("G"),}
\text{ConcatTok(Interval(NumChar, 4), UpperChar))}
\text{)})
\]

Figure 1: Example from help forum. Shows original NL task description, examples and synthesized program.

2 Compositional Synthesis Framework

Domain Specific Language (DSL). The DSL is the language within which programs will be synthesized. It is defined as a context-free grammar of the form \((\psi_\text{NT}, \psi_T, \psi_\text{start}, \text{Rules})\), where \(\psi_\text{NT}\) is a set of non-terminal symbols, \(\psi_T\) is the set of terminal symbols, \(\psi_\text{start}\) is the start symbol and \(\text{Rules}\) is the set of non-terminal production rules of the grammar. Each production rule \(\text{rule} \in \text{Rules}\) is of the form \((\text{ruleName}, \psi_h, \text{Body})\) where \(\text{ruleName}\) is the rule name, \(\psi_h \in \psi_\text{NT}\) is the head symbol, and \(\text{Body}\) is a sequence of symbols \((\psi_1, \ldots, \psi_n)\) where each \(\psi_i \in \psi_\text{NT} \cup \psi_T\). The semantics of the DSL is given by an interpretation of every symbol \(\psi\) as ranging over a set of values \([\psi]\), and an interpretation of each rule \(\text{rule}\) as a function

\[
[\text{rule}] : [\psi_1] \times \ldots \times [\psi_n] \rightarrow [\psi_h]
\]

where \(\text{rule}.\text{Body} = (\psi_1, \ldots, \psi_n)\). A program \(P\) of type \(\psi\) is any concrete syntax tree defined by the DSL grammar with root symbol \(\psi\). A complete program is a program with root symbol \(\psi_\text{start}\). Any derivation from a non-root symbol is an incomplete program or a program component.

Figure 2 illustrates a particular instantiation DSLs for string transformations that we use in this paper. The start symbol of the language is \(f\) which ranges over strings. The non-terminal symbols are given on the left hand side on each line along with their semantic value ranges in bold. The terminal symbols of the language are \(k, n, c, s\) as well as the special symbol \(\text{input}\) which represents the input string on which a program executes. The \(\text{input}\) symbol is the first parameter for every rule, but is omitted in the figure for brevity.

The DSL includes the language of Flash Fill [Gulwani,
\[
\begin{align*}
\text{string } f & := \text{SubStr}(p, p) | \text{SubStr2}(r, i) | \text{ConstStr}(s) | \text{ConstChar}(c) | \text{Filter}(r) | \text{Replace}(r, s) | \text{Remove}(p) | \\
& \quad | \text{Loop}(w, f) | \text{end} | \text{ Concat}(f, f) \\
\text{string } end & := \text{IfThen}(b, f) | \text{IfThenElse}(b, f, f) \\
\text{bool } b & := \text{Match}(r, n) | \text{Not}(b) | \text{And}(b, b) | \text{Or}(b, b) \\
\text{int } p & := \text{CPos}(k) | \text{Pos}(r, i, r) \\
\text{rec } r & := \text{EmptyTok} | \text{StartTok} | \text{EndTok} | \text{StrTok}(s) | \text{CharTok}(c) | \text{chCl} | \text{Neg}(\text{chCl}) | \\
& \quad | \text{ConcatTok}(r, r) | \text{Interval}(r, n, n) | \text{Interval}(r, n) | \text{Optional}(r) | \text{KleenePlus}(r) | \text{KleeneStar}(r) | \\
& \quad | \text{DisjTok}(r, r) | \text{LkBehind}(r) | \text{LkAhead}(r) | \text{IncludeWS}(r) \\
\text{rec } \text{chCl} & := \text{AnyChar} | \text{NumChar} | \text{LowerChar} | \text{UpperChar} | \text{LiteralChar}(c) | \text{Union}(\text{chCl}, \text{chCl}) \\
\text{int } i & := k \mid k * w + k
\end{align*}
\]

Figure 2: DSLs for string transformations. The start symbol is \( f \), the terminal symbols \( k, n, c \) and \( s \) represent literal values of type integer, natural number, character and string respectively, and \( w \) is an integer variable.

2011], but the strong expressivity limitations required in that work are lifted to permit unrestricted regular expression operators such as disjunction, iteration, negation, look-arounds (within arbitrary levels of nesting), as well as additional string transformation operators such as replace, remove, and filter. This lifting of the expressivity limitations is one of the features permitted by the compositional synthesis approach which we evaluate here.

We next discuss briefly the semantics of the DSL operators (for detailed semantics of Flash Fill operators see [Gulwani, 2011]). The \( \text{SubStr}(p_1, p_2) \) operator extracts the substring between the positions \( p_1 \) and \( p_2 \) in the input string, while \( \text{SubStr2}(r, i) \) extracts the \( i \)th occurrence of regular expression \( r \) from the input string. \( \text{ConstStr} \) and \( \text{ConstChar} \) represent constant string and character values. \( \text{Filter}(r) \) returns the input string if it satisfies regex \( r \) and null otherwise. \( \text{Replace}(r, s) \) replaces every occurrence of a string matching \( r \) in the input with string \( s \). \( \text{Remove}(p) \) removes everything after position \( p \) in the input. \( \text{Concat}(f, f) \) returns the concatenation of the two strings. \( \text{IfThen}(b, f) \) returns \( f \) if \( b \) is true and the input string otherwise, while \( \text{IfThenElse}(b, f_1, f_2) \) returns \( f_2 \) in the else case. The \( \text{Loop}(w, f) \) operator produces a concatenation of strings that are instances of \( f \), where the \( k \)th instance is the substitution of \( w = k \) for all occurrences of the variable \( w \) in \( f \). Boolean conditions \( b \) (based on the \( \text{Match}(r, n, r) \) predicate which asserts that there are \( n \) occurrences of regex \( r \) in the input string. Positions \( p \) in the input string are generated by either the constant position constructor \( \text{CPos}(k) \), or \( \text{Pos}(r_1, r_2, i) \) which is the \( i \)th occurrence of a position in the input where the left satisfies \( r_1 \) and the right satisfies \( r_2 \).

Regular expressions \( r \) include standard regex operators. There are character classes \( \text{chCl} \) for any character, numeric, lower case, upper case, literal characters and unions of classes. Tokens include empty, start, end, literals, character classes and their negations \( \text{Neg}(\text{chCl}) \). \( \text{ConcatTok}(r, r) \) concatenates two tokens. Iteration operators include \( \text{Interval}(r, n_1, n_2) \) (at least \( n_1 \) and at most \( n_2 \) occurrences of \( r \)), \( \text{Interval}(r, n) \) (exactly \( n \) occurrences), \( \text{Optional}(r) \) (zero or one), \( \text{KleenePlus}(r) \) (at least one) and \( \text{KleeneStar}(r) \) (zero or more). \( \text{DisjTok}, \text{LkBehind} \) and \( \text{LkAhead} \) implement alternation (disjunction), look-behind and look-ahead. IncludeWS \( (r) \) matches the regex \( r \) including any surrounding whitespace on either side. Regex semantics is given by occurrence records \( \text{rec} = \mathcal{P}(\text{int} \times \text{int}) \). An occurrence record \( \rho \in \text{rec} \) is a set of pairs of start and end indexes representing all possible matches of the regex in the input string. This semantics provides a strong observational equivalence relation between regexes over a given set of examples, which helps to significantly reduce the search space in the synthesis.

**Compositional Specifications.** Let \( \Sigma \) be the domain of objects e.g., the set of strings for string manipulation tasks. A standard input-output examples specification of a task is usually a pair \( (\phi_1, \phi_2) \) such that \( \phi_1, \phi_2 \in \Sigma^n \), specifying \( n \) input states and their corresponding outputs. In our case, we define the notion of a compositional examples specification which, in addition to the input and output examples, also specifies examples of constituent states. A compositional specification with \( n \) input examples is defined as \( \phi = (\phi_1, \phi_2) \) where \( \phi_1 \in \Sigma^n \) and \( \phi_2 \) is a tree \( t \in T \) of the form \( t := \hat{e}[t_1, t_2, \ldots, t_l] \). A tree node \( e \in \mathcal{P}(\Sigma)^n \) is an \( n \)-tuple of sets of examples. We permit a set of examples in the output nodes to allow multiple examples to be given for constituent concepts, e.g. for the task “allow only letters and numbers”, the user may give multiple examples of “letters” occurring in the input string. The root node of the tree \( \phi_2 \) specifies the final output for each input, and should therefore always be a tuple of singleton sets.

**Example 1** For the NL task description “Any 2 letters” followed by any combination of “6 whole numbers”, the user provides one positive and one negative example for the input, output and constituent concepts:

<table>
<thead>
<tr>
<th></th>
<th>Ex 1</th>
<th>Ex 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>RJ123456</td>
<td>DDD12345</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>RJ123456</td>
<td>null</td>
</tr>
<tr>
<td><strong>&quot;Any 2 letters&quot;</strong></td>
<td>RJ</td>
<td></td>
</tr>
<tr>
<td><strong>&quot;6 whole numbers&quot;</strong></td>
<td>123456</td>
<td></td>
</tr>
</tbody>
</table>

These examples are represented by the compositional spec \( (\phi_1, \phi_2) \) where \( \phi_1 = (\"RJ123456\", \"DDDJ2345\") \) and \( \phi_2 = \hat{e}_1[\hat{e}_2, \hat{e}_3] \) where \( \hat{e}_1 = (\{\"RJ123456\\}, \{\text{null}\}), \hat{e}_2 = (\{\"RJ\\}, \emptyset) \) and \( \hat{e}_3 = (\{\"123456\\}, \emptyset) \).

The tree structure of the output also permits constituent ex-

amples to be given for every node, which may be required for more complex constituent concepts. We refer to the root node in the output tree as the output examples node and every other node as a constituent-examples node. Intuitively, a given program \( P \) in the DSL satisfies a compositional spec \( \phi \) if it satisfies the input and output examples, and is composed of components that “satisfy” the constituent-examples nodes, where this notion of satisfaction is formalised by a component satisfaction relation which we define next.

**Component Satisfaction Relation (CSR).** When the user gives a set of constituent examples, the examples could be referring to any component program of a certain type in the DSL, whether it is a regular expression, a character class, a position expression, etc. Each of these types have their own semantic values, and the CSR is meant to describe the relationship between the given examples and values of this type: the values that may be relevant for the given examples. For instance, for a given set of string examples, any occurrence records matching those examples may be relevant regular expression values. Formally, for each non-terminal symbol \( \psi \in \psi_{NT} \) in the DSL, a separate relation \( CSR(\psi) \) is defined. Assume we are given input examples \( \hat{\phi}_1 = (e_1, ..., e_n) \) and a constituent-examples node \( \hat{e} \). Let \( \bar{v} = (v_1, ..., v_n) \) be a tuple of \( n \) values such that \( v_i \in [\psi] \) (values may be generated by a program component operating on the input states \( \hat{\phi}_1 \)). The relation \( CSR(\psi)(\hat{\phi}_1, \hat{e}, \bar{v}) \) determines whether the values correspond to the constituent examples on the given input states. The definition of the CSR relation is a design choice that is up to the DSL designer, to specify how language components would relate to a given set of examples. The only constraint required is that the CSR for the start symbol should be exactly the semantics of complete programs in the DSL, that is, the value tuple should correspond exactly to the output examples.

For instance, the CSR relations we have defined for the string DSL are given in Figure 3. For the start symbol \( \psi \) which represents complete programs, the values correspond exactly to the examples. For character classes, we require all the characters in the given examples and value tuple to fall under the same class. For example, in the case of \( \hat{\phi}_1 \) and \( \hat{e}_2 \) from Example 1, any value tuple of occurrence records including non-capital letters will not satisfy the CSR, since the examples only includes capital letters. For regular expressions \( \hat{r} \), we require all the example strings to be included in the matches given in the occurrence records in the value tuple. Hence in the case of \( \hat{e}_2 \) in Example 1, the occurrence records generated by KleenePlus(UpperChar) and Interval(UpperChar, 2) will satisfy the CSR, but Interval(UpperChar, 1) will not. For position expressions, we require the example strings to occur in the input string at either the start or end positions that are given in the value tuple. So in the case of \( \hat{e}_2 \), valid value tuples will contain the positions 0 or 2, which may for example be generated by CPos(0), CPos(2) or Pos(UpperChar, NumChar, 0). For other non-terminals, we define the CSR to be false, to indicate no direct relationship between examples and components of this type.

Apart from the relations defined for the non-terminal symbols, for every terminal symbol \( \psi \in \psi_T \) the CSR defines a set of literal values to be used for these terminals: \( CSR(\psi) \subseteq \{v_i \} \) if \( v_i \in \Sigma \).

---

**3 Program Synthesis Algorithm**

In this section we describe the program synthesis algorithm which is parametric in a given DSL, CSR and a compositional specification. At its core, the algorithm performs a systematic search over the state space of possible programs, but this search is optimized by incorporating components that are recursively synthesized from the compositional specification. These components are also used to rank among numerous satisfying programs that may be generated from the search. This combination of systematic and specification-guided heuristic techniques means that the algorithm is theoretically sound and complete (it will always generate a program satisfying the input-output examples if one exists, even without any constituent examples), and also performs efficiently on complex tasks in practice, as we describe in the evaluation. The main procedures of the algorithm are shown in Figure 4. In the following description we assume a given DSL, CSR and specification \( \phi = (\hat{\phi}_1, \hat{\phi}_2) \) with a fixed number of examples \( n \).

**Value maps.** The algorithm uses value maps to efficiently maintain large sets of programs that yield the same values (are observationally equivalent) on the given input examples in \( \phi \). A value map \( \theta \) is a partial map \( \theta[\psi, \bar{v}] = \bar{P} \) which maps a DSL symbol \( \psi \) and a value tuple \( \bar{v} \in [\psi]^n \) to a set of programs \( \bar{P} \) of type \( \psi \). We denote \( dom(\theta) \) for the set of pairs \((\psi, \bar{v})\) in the domain of \( \theta \), and \( Symbols(\theta) \) for the set of symbols in the domain of \( \theta \). We write \( \theta[\psi] \) to indicate the restriction of the domain to just \( \psi \). We define the union of maps \( \theta_1 \cup \theta_2 = \theta \) such that \( \theta[\psi, \bar{v}] = \theta_1[\psi, \bar{v}] \cup \theta_2[\psi, \bar{v}] \) if \( (\psi, \bar{v}) \in dom(\theta_1) \land (\psi, \bar{v}) \in dom(\theta_2) \).

We next define application of DSL rules to value maps. Let \( rule = (ruleName, \psi_{\theta}, (v_1, ..., v_m)) \) be a DSL rule. For \( 1 \leq i \leq m \), let \( \bar{v}_i = [\psi_{\theta}]^n \) such that \( \bar{v}_i = (v_{i1}, ..., v_{in}) \). The lifting of rule application to value tuples is defined as \([rule](v_1, ..., v_m) = \bar{v} \) such that \( \bar{v} = (v_1, ..., v_m) \) and for \( 1 \leq k \leq n \) we have \( v_{ik} = [rule](v_{i1}, ..., v_{im}) \). Rule application for value maps is defined as \([rule](\theta_1, ..., \theta_m) = \theta \).

---

**Figure 3: CSR(\psi) for DSLs.** We let \( \hat{\phi}_1 = (s_1, ..., s_n) \), \( \hat{e} = (e_1, ..., e_n) \) and \( \bar{v} = (v_1, ..., v_n) \) with \( v_i \in [\psi] \). We initialise these sets with literal values that occur in the natural language task description. Hence for Example 1, the numeric values 2 and 6 are used as integer literals, and for the example in Figure 1, “G” is used as a literal character.
1: function SynthProgram(DSL, CSR, φ)
2:    ψ := start symbol of DSL
3:    ˆe := root node of φ
4:    θ := SynthCSRStates(DSL, CSR, ˆe, ψ)
5:    return GetTopRankedProgram(φ, θ)

1: function SynthCSRStates(DSL, CSR, φ, ˆe, ψ)
2:    θI := GetCSRTerminalValuesMap(CSR, φ)
3:    ψNT := non-terminal symbols of DSL
4:    Rules := rules of DSL
5:    let θ map constituent-examples nodes to value maps
6:    for each child of ˆe do
7:       M[e] := SynthCSRStates(DSL, CSR, ˆe, ψNT)
8:       θI := θI ∪ M[e]
9:    θI := θI ∪ AggregatorRules(DSL, M)
10:   θI := θI ∪ ModifierRules(DSL, M)
11:   θR := GetCSRValuesMap(CSR, θI, φ, ψ)
12:   ψ := ψ − Symbols(θR)
13:   while ψ ≠ ∅ do
14:      θCur := θI ∪ θR
15:      θCur := ∅
16:   while θCur ≠ ∅ do
17:      θCur := θCur ∪ ApplyRules(Rules, θCur)
18:   θR := GetCSRValuesMap(CSR, θCur, φ, ψ)
19:   ψ := ψ − Symbols(θR)
20:   return θR

Figure 4: Program synthesis algorithm

such that θ[ψ, v] = ˆP if and only if there exist ˜v1, ..., ˜vm such that θ[ψ, ˜v1] = ˆP1, ˜v = [rule](ψ1, ..., ψm) and

\[ ˆP = \{ P \mid P = ruleName(P1, ..., Pm) \land P1 \in ˆP1 \} \]

We write \[ [rule](θ) = [rule](θ1, ..., θm) \] if θi = θ for all i. For a rule set Rules, we define

\[ \text{ApplyRules}(Rules, θ) = \bigcup_{rule ∈ Rules} [rule](θ) \]

Main algorithm. The main function of the algorithm is SynthProgram defined in Figure 4, which takes a DSL, CSR and compositional spec and returns a program. This function first generates a value map of satisfying programs by calling the recursive function SynthCSRStates with the root node of the specification and the start symbol of the grammar. It then returns the top ranked program according to a specification-based ranking scheme which we describe below.

The SynthCSRStates function takes a DSL, CSR, a specification φ, a constituent-examples node ˆe from φ, and a set of symbols ψ. It returns a value map which, for each symbol ψ ∈ ψ, contains programs of type ψ that satisfy the CSR(ψ) with respect to the examples node ˆe. Hence when called with the root node of the specification and the start symbol (in the main function), it returns a set of complete programs satisfying the input-output examples. The SynthCSRStates function can be described in two main phases: the initialization phase (lines 1-10) is the generation of an initial set of program components in value map θI. These initial components are then used in the search phase (lines 11 to 21) to perform a systematic search through the space of possible programs by iteratively generating increasingly bigger programs, similar in style to [Katayama, 2007].

Initialization. The value map θI is first initialized with all the terminal components specified in the CSR terminal state sets. This is done by the GetCSRTerminalValuesMap function:

\[ \text{GetCSRTerminalValuesMap}(CSR, φ) \]

return θI such that θI[ψ, v] = v for all terminal symbols ψ, v ∈ CSR(ψ) and v = (v1, ..., vn) of size n.

Then, for each child of the node ˆe, there is a recursive call to generate components satisfying the child node and these components are added to θI (lines 3-8). The algorithm then gives priority to two commonly occurring rule application patterns over the constituent components, before going into brute-force search. The first is for the aggre- gator rules, which are binary recursive rules of the form \( ruleName(ψ1, ψ2) \). Examples of such rules in the string DSL are Concat, DisjTok, ConcatTok. And, Or and in other DSLs may include operators such as sequential composition. Such recursive rules are often used to aggregate components together by repetitive application of the form \( ruleName(P1, ..., Pn) \), e.g. tasks requiring a sequence of concatenations or disjunctions. For each aggregator rule \( rule = (ruleName, ψ1, ψ2) \) in the DSL and sub-sequence of child nodes ˆe1, ..., ˆej, the function AggregatorRules performs the rule application \( [rule](M[ε1], ..., M[εj]) \).

Similarly, the modifier rule pattern applies unary recursive rules of the form \( ruleName(ψ) \). Examples of such rules may be KleeneStar, LkBehind or Not. Constituent concepts in task specifications are often modified with such rules when used in the full program. For example, in the specification “extract all characters that occur after a number”, the regular expression for “number” may be used under the application of a look-behind operator. For each modifier rule \( rule \) in the DSL and child node ˆe, the function ModifierRules performs the rule application \( [rule](M[ε]) \).

Search. The search phase begins at line 11, where it is first checked if any CSR-satisfying components have already been generated:

\[ \text{GetCSRValuesMap}(CSR, φ, ψ) \]

return θR such that θR[ψ, v] = v if ψ ∈ CSR(ψ) and v ∈ CSR(ψ) and undefined otherwise.

The value map θR collects CSR states for all the required symbols. Using the initial components from θI, rules of the DSL are iteratively applied until CSR-satisfying components have been found for all the required symbols (lines 13-20). In practice we apply timeouts for rule application and recursive calls for component synthesis.

Ranking. The function GetTopRankedProgram ranks among satisfying programs using a relation that is a lexical ordering of three metrics \( CSRScore, Size, NumCSRComps \). For a program \( P \), \( CSRScore(P) \) is the number of constituent example nodes for which \( P \) contains a satisfying component + the number of literal terminal values from the CSR relation that occur in \( P \). The \( Size(P) \) is the number of nodes in the syntax tree of \( P \). \( NumCSRComps(P) \) is the total number
of components in $P$ that satisfy the CSR. Hence the ranking scheme is to first prefer programs that satisfy the most constituent examples nodes and CSR terminal values, then to choose the smallest from among these, and then from these choose the one with the most CSR satisfying components.

4 Evaluation

We evaluated the system on a set of tasks from online help forums which are covered by the wide range of constructs provided by DSLs, including conditionals, loops, the various string transformation operators and complex regular expressions. For each task, we used the natural language description as stated in the original forum question, and obtained the noun phrases for constituent concepts from this description using the Stanford [Klein and Manning, 2003] and SPLAT constituency parsers [Quirk et al., 2012].

Out of the 48 tasks, the system synthesized correct programs for 42 and timed out on the remaining 6. The average number of examples required was 2.73, with a maximum of 6 examples for one of the tasks. The average number of constituent concepts required per task was 1.53, with a maximum of 4 and minimum of 0. The average execution time was 9.97 seconds, with 28 tasks completing in under 4 seconds.

In addition to the full system, we compared against three baselines as shown in Figure 5. The first was the Flash Fill (FF) system [Gulwani, 2011], which uses a domain-specific algorithm for a subset of the string manipulation DSL we use here. FF gave correct results on 2 of the 48 tasks (only 8 were expressible in the FF language and 6 of those yielded incorrect programs). For the second baseline B1, we supplied our system with only the input-output examples for each task and no constituent examples. Only 7 programs were synthesized correctly in this case, demonstrating the improvement achieved with compositionality. For the third baseline B2, we applied our system with a ranking scheme that chose the smallest program as in [Gulwani, 2011]. In this case 35 programs were correctly synthesized, showing the role of compositionality not only in the tractability of search, but in the accuracy of ranking as well. Below are some sample fragments of NL descriptions for tasks handled by our system (we will also illustrate the synthesized programs and other details in extra pages in the final version of the paper).

- **first character** must be a letter (“A”, “B”, or “C” only) the remaining 4 characters can be number or letter.
- If the cells contain a 16 digit number then -Replace the first 12 digits of each string with “xxxxXXXxxxx”
- extract any numbers after “SN”. the numbers can be vary in digits. Also, at times there is some other text in between numbers and search word
- The string must start with “1” or “2” (only once and mandatory) and then followed by any character between “a” to “z” (only once)
- If column A contains the words “ear” or “mouth”, then I want to return the value of “face” otherwise I want it to return the value of “body”
- a digit, following by the letter “m” or “M”, followed by a digit, then followed by the letter “f” or “F”.
- first 3 character is alphabet (lower and upper both) then any alphanumeric if present

<table>
<thead>
<tr>
<th></th>
<th>FF</th>
<th>B1</th>
<th>B2</th>
<th>CPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of timeouts</td>
<td>0</td>
<td>26</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Number of incorrect results</td>
<td>46</td>
<td>15</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Number of correct results</td>
<td>2</td>
<td>7</td>
<td>35</td>
<td>42</td>
</tr>
<tr>
<td>Average time (seconds)</td>
<td>&lt; 0.5</td>
<td>12.35</td>
<td>8.99</td>
<td>9.97</td>
</tr>
</tbody>
</table>

Figure 5: Baselines (FF, B1,B2) and full system (CPS) on 48 tasks

5 Related Work and Conclusion

In recent years there has been much work in the area of program synthesis from natural language, examples or a mixture of such approaches. Natural language learning approaches have addressed the translation of sentences to meaning representations such as database queries [Zettlemoyer and Collins, 2009; Clarke et al., 2010; Gulwani and Marron, 2014; Liang et al., 2011], navigation plans [Chen and Mooney, 2011] and string manipulation expressions [Manshadi et al., 2013; Kushman and Barzilay, 2013]. Apart from ambiguity issues in NL, such approaches are limited by the adequacy of the domain-specific training phase. Although we have demonstrated the compositional examples-based approach independently of any language learning supervision, the incorporation of such advanced NLP techniques is expected to reduce the amount of constituent examples required, while still supporting complex tasks when language understanding fails.

There has also been significant work on purely example-based approaches for string manipulation tasks [Gulwani, 2011; Raza et al., 2014; Le and Gulwani, 2014; Perelman et al., 2014]. However, all of these approaches impose strong limitations on the expressivity of the DSL, which can be avoided with the compositional approach as we demonstrate in this work. The closely related area of Programming-by-Demonstration [Lau et al., 2003] also advocates a degree of compositionality, as users can demonstrate traces of actions rather than just give input-output examples. However, this is only true for the concrete actions that can be performed rather than expressing the general conditions under which the action is to be performed e.g. a complex regular expression for extracting a string (such as in Figure 1) cannot be expressed through a series of actions. It will be beneficial to incorporate such approaches when the user has knowledge of the demonstrational interface, which may be true in particular domains such as web browsing [Allen et al., 2007] but not in general.

In summary, we have described a domain-agnostic program synthesis framework and algorithm, with which complex tasks can be accomplished by providing input in a compositional manner. In particular, we have demonstrated the system in the particular domain of string manipulation, supporting a very expressive domain language and evaluating with complex examples from online help forums. In the long run, as we attempt to address more sophisticated tasks, one can imagine moving towards a dialog-based interaction model. For example, much like the experts on help forums, the system may request the user for elaboration or examples of concepts mentioned, present paraphrased natural language descriptions of synthesized programs to the user, and request counter-examples if such proposals are not correct. We aim to explore such interactive approaches in future work.
References


