Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Compositional Program Synthesis from Natural Language and Examples

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

Sumit Gulwani
Microsoft Research
Redmond, USA
sumitg@microsoft.com

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

Abstract

Compositionality is a fundamental notion in computation whereby complex abstractions can be constructed from simpler ones, yet 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 specifications of tasks, which limits the expressivity and scalability of these approaches to relatively simple programs in very restricted domains. In this paper we propose Compositional Program Synthesis (CPS): an approach in which tasks can be specified 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 our approach on complex tasks from online help forums that are beyond the scope of current state-of-the-art methods.

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 such informal descriptions of tasks into a computer program expressed in an underlying domain specific language (DSL) that is unknown to 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 (as 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 a very restricted subset of regular expressions (without literal string tokens, disjunctions, or iterations) and therefore disallows 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] can potentially support more expressive DSLs, as they enjoy a stronger preference bias coming from explicit natural language descriptions of intent as opposed to examples only. 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 issues, 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, whether it is an NL description or input-output examples describing the whole task at once. In contrast, compositionality is at the heart of programming practice, with fundamental abstractions such as expressions, procedures, classes or libraries providing modularity and separation of concerns that helps construct complex programs 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 correct answer.

Hence as tasks get more sophisticated, some kind of compositional paradigm is the way forward. But one reason this is difficult to achieve in end user settings is that the user has no knowledge 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 concepts 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 [Manshadi *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 that supports unrestricted regular expressions, there could be many possible programs that satisfy the 4 given examples. However, using the intermediate examples of the constituent concepts, the system first generates relevant components such as the interval expression $\mathsf{Interval}(\mathsf{NumChar}, 1, 5)$ for the concept "1-5 numbers", $\mathsf{Interval}(\mathsf{NumChar}, 4)$ for "4 numbers" and the single character token $\mathsf{UpperChar}$ for "a single letter". Using these relevant components, 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 applying 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" followed by **1-5 numbers** or "G" followed by **4 numbers** followed by **a single letter** "A"-"Z"*

Examples:

	Ex 1	Ex 2	Ex 3	Ex 4
Input	G2	G12345	G1234B	G123456
Output	G2	G12345	G1234B	null
"1-5 numbers"	2	12345		
"4 numbers"			1234	
"a single letter"			B	

Synthesized program:

```
Filter(
  DisjTok(
    ConcatTok(
      CharTok('G'),
      Interval(NumChar, 1, 5)),
    ConcatTok(
      CharTok('G'),
      ConcatTok(Interval(NumChar, 4), UpperChar))
  )
)
```

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 $(\tilde{\psi}_{NT}, \tilde{\psi}_T, \psi_{start}, Rules)$, where $\tilde{\psi}_{NT}$ is a set of non-terminal symbols, $\tilde{\psi}_T$ is the set of terminal symbols, ψ_{start} is the start symbol and $Rules$ is the set of non-terminal production rules of the grammar. Each production rule $rule \in Rules$ is of the form $(ruleName, \psi_h, Body)$ where $ruleName$ is the rule name, $\psi_h \in \tilde{\psi}_{NT}$ is the head symbol, and $Body$ is a sequence of symbols $(\psi_1, \ldots, \psi_n)$ where each $\psi_i \in \tilde{\psi}_{NT} \cup \tilde{\psi}_T$. The semantics of the DSL is given by an interpretation of every symbol ψ as ranging over a set of values $[\![\psi]\!]$, and an interpretation of each rule $rule$ as a function

$$[\![rule]\!] : [\![\psi_1]\!] \times, \ldots, \times [\![\psi_n]\!] \rightarrow [\![\psi_h]\!]$$

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

Figure 2 illustrates a particular instantiation DSL_s 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 $input$ which represents the input string on which a program executes. The $input$ 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, 2011], but we lift the strong expressivity restrictions required in that work and permit unrestricted regular expression op-

$$
\begin{aligned}
\textbf{string } f &:= \mathsf{SubStr}(p,p) \mid \mathsf{SubStr2}(r,i) \mid \mathsf{ConstStr}(s) \mid \mathsf{ConstChar}(c) \mid \mathsf{Filter}(r) \mid \mathsf{Replace}(r,s) \mid \mathsf{Remove}(p) \mid \\
&\quad \mathsf{Loop}(\mathsf{w},f) \mid cnd \mid \mathsf{Concat}(f,f) \\
\textbf{string } cnd &:= \mathsf{IfThen}(b,f) \mid \mathsf{IfThenElse}(b,f,f) \\
\textbf{bool } b &:= \mathsf{Match}(r,n) \mid \mathsf{Not}(b) \mid \mathsf{And}(b,b) \mid \mathsf{Or}(b,b) \\
\textbf{int } p &:= \mathsf{CPos}(k) \mid \mathsf{Pos}(r,r,i) \\
\textbf{rec } r &:= \mathsf{EmptyTok} \mid \mathsf{StartTok} \mid \mathsf{EndTok} \mid \mathsf{StrTok}(s) \mid \mathsf{CharTok}(c) \mid chCl \mid \mathsf{Neg}(chCl) \mid \\
&\quad \mathsf{ConcatTok}(r,r) \mid \mathsf{Interval}(r,n,n) \mid \mathsf{Interval}(r,n) \mid \mathsf{Optional}(r) \mid \mathsf{KleenePlus}(r) \mid \mathsf{KleeneStar}(r) \mid \\
&\quad \mathsf{DisjTok}(r,r) \mid \mathsf{LkBehind}(r) \mid \mathsf{LkAhead}(r) \mid \mathsf{IncludeWS}(r) \\
\textbf{rec } chCl &:= \mathsf{AnyChar} \mid \mathsf{NumChar} \mid \mathsf{LowerChar} \mid \mathsf{UpperChar} \mid \mathsf{LiteralChar}(c) \mid \mathsf{Union}(chCl, chCl) \\
\textbf{int } i &:= k \mid k * \mathsf{w} + k
\end{aligned}
$$

Figure 2: DSL_s 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.

erators such as disjunction, iteration, negation, look-arounds, arbitrary literals, as well as additional string transformation operators such as replace, remove, and filter. This lifting of the expressivity limitations is one of the key features permitted by the compositional synthesis approach that we evaluate here.

We next discuss briefly the semantics of the DSL operators (for detailed semantics of Flash Fill operators see [Gulwani, 2011]). The $\mathsf{SubStr}(p_1, p_2)$ operator extracts the substring between the positions p_1 and p_2 in the input string, while $\mathsf{SubStr2}(r, i)$ extracts the i^{th} occurrence of regular expression r from the input string. ConstStr and ConstChar represent constant string and character values. $\mathsf{Filter}(r)$ returns the input string if it satisfies regex r and null otherwise. $\mathsf{Replace}(r, s)$ replaces every occurrence of a string matching r in the input with string s. $\mathsf{Remove}(p)$ removes everything after position p in the input. $\mathsf{Concat}(f, f)$ returns the concatenation of the two strings. $\mathsf{IfThen}(b, f)$ returns f if b is true and the input string otherwise, while $\mathsf{IfThenElse}(b, f_1, f_2)$ returns f_2 in the else case. The $\mathsf{Loop}(\mathsf{w}, f)$ operator produces a concatenation of strings that are instances of f, where the k^{th} instance is the substitution of $\mathsf{w} = k$ for all occurrences of the variable w in f. Boolean conditions (b) are based on the $\mathsf{Match}(r, n)$ 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 $\mathsf{CPos}(k)$, or $\mathsf{Pos}(r_1, r_2, i)$ 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 ($chCl$) 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 $\mathsf{Neg}(chCl)$. $\mathsf{ConcatTok}(r, r)$ concatenates two tokens. Iteration operators include $\mathsf{Interval}(r, n_1, n_2)$ (at least n_1 and at most n_2 occurrences of r), $\mathsf{Interval}(r, n)$ (exactly n occurrences), Optional (zero or one), KleenePlus (at least one) and KleeneStar (zero or more). DisjTok, LkBehind and LkAhead implement alternation (disjunction), look-behind and look-ahead. $\mathsf{IncludeWS}(r)$ matches the regex r including any surrounding whitespace on either side. Regex semantics is given by *occurrence records* $\mathbf{rec} = \mathcal{P}(\mathbf{int} \times \mathbf{int})$. An occurrence record $\rho \in \mathbf{rec}$ 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 Σ 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 (ϕ_I, ϕ_O) such that $\phi_I, \phi_O \in \Sigma^n$, 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 $\phi = (\phi_I, \phi_O)$ where $\phi_I \in \Sigma^n$ and ϕ_O is a tree $t \in T$ of the form $t := \hat{e}[t, \ldots, t]$. A tree node $\hat{e} \in \mathcal{P}(\Sigma)^n$ 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 ϕ_O 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:*

	Ex 1	*Ex 2*
Input	*RJ123456*	*DDD12345*
Output	*RJ123456*	*null*
"Any 2 letters"	*RJ*	
"6 whole numbers"	*123456*	

These examples are represented by the compositional spec (ϕ_I, ϕ_O) *where* $\phi_I = (\text{"RJ123456"}, \text{"DDD12345"})$ *and* $\phi_O = \hat{e}_1[\hat{e}_2, \hat{e}_3]$ *where* $\hat{e}_1 = (\{\text{"RJ123456"}\}, \{null\})$, $\hat{e}_2 = (\{\text{"RJ"}\}, \emptyset)$ *and* $\hat{e}_3 = (\{\text{"123456"}\}, \emptyset)$.

The tree structure of the output also permits constituent examples 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 ϕ 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 $\psi \in \tilde{\psi}_{NT}$ in the DSL, a separate relation $CSR\langle\psi\rangle$ is defined. Assume we are given input examples $\phi_I = (e_1, ..., e_n)$ and a constituent-examples node $\hat{e}$. Let $\bar{v} = (v_1, \ldots, v_n)$ 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 ϕ_I). The relation $CSR\langle\psi\rangle(\phi_I, \hat{e}, v)$ 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 f 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 ϕ_I and $\hat{e}_2$ from Example 1, any value tuple of occurrence records including non-capital letters will not satisfy the CSR, since the example only includes capital letters. For regular expressions (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 $\mathsf{KleenePlus}(\mathsf{UpperChar})$ and $\mathsf{Interval}(\mathsf{UpperChar}, 2)$ will satisfy the CSR, but $\mathsf{Interval}(\mathsf{UpperChar}, 1)$ 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 $\mathsf{CPos}(0)$, $\mathsf{CPos}(2)$ or $\mathsf{Pos}(\mathsf{UpperChar}, \mathsf{NumChar}, 0)$. 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 $\psi \in \tilde{\psi}_T$ the CSR defines a set of literal values to be used for these terminals: $CSR\langle\psi\rangle \subseteq [\![\psi]\!]$. 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.

$$
\begin{array}{rll}
CSR\langle f\rangle(\phi_I, \hat{e}, \bar{v}) & \text{iff} & \forall i.\ e_i = \{v_i\} \\
CSR\langle chCl\rangle(\phi_I, \hat{e}, \bar{v}) & \text{iff} & Chars(\cup_i Strings(s_i, v_i)) \text{ and } Chars(\cup_i e_i) \text{ belong to the same minimal character class} \\
CSR\langle r\rangle(\phi_I, \hat{e}, \bar{v}) & \text{iff} & \forall i. e_i \subseteq Strings(s_i, v_i) \text{ and } CSR\langle chCl\rangle(\phi_I, \hat{e}, \bar{v}) \\
CSR\langle p\rangle(\phi_I, \hat{e}, \bar{v}) & \text{iff} & \forall i. \forall e \in e_i.\ e \text{ is a substring of } s_i \text{ which has either start or end position } v_i \text{ in } s_i
\end{array}
$$

$Strings(s, \rho)$ = all substrings of string s that are matches determined by occurrence record ρ

$Chars(S)$ = all characters occurring in any string in the set S

Figure 3: $CSR\langle\psi\rangle$ for DSL_s. We let $\phi_I = (s_1, ..., s_n)$, $\hat{e} = (e_1, ..., e_n)$ and $\bar{v} = (v_1, ..., v_n)$ with $v_i \in [\![\psi]\!]$

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 performs efficiently on complex tasks in practice, but is also theoretically sound and complete in the sense that if the required terminal symbols are given in the CSR and no timeouts are imposed then it will always generate a program satisfying the input-output examples if one exists (even if no constituent examples are given).

In the rest of the description in this section we assume a given DSL, CSR and specification $\phi = (\phi_I, \phi_O)$ with a fixed number of examples n. Before describing the main algorithm, we first give some preliminary definitions.

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 ϕ. A value map θ is a partial map $\theta[\psi, \bar{v}] = \tilde{P}$ which maps a DSL symbol ψ and a value tuple $\bar{v} \in [\![\psi]\!]^n$ to a set of programs $\tilde{P}$ of type ψ. We write $dom(\theta)$ for the set of pairs $(\psi, \bar{v})$ in the domain of θ, and $Symbols(\theta)$ for the set of symbols in the domain of θ. We write $\theta|_\psi$ to indicate the restriction of the domain to just ψ. We define the union of maps $\theta_1 \cup \theta_2 = \theta$ such that $\theta[\psi, \bar{v}] =$

$$
\begin{cases}
\theta_1[\psi, \bar{v}] \cup \theta_2[\psi, \bar{v}] & (\psi, \bar{v}) \in dom(\theta_1) \wedge (\psi, \bar{v}) \in dom(\theta_2) \\
\theta_1[\psi, \bar{v}] & (\psi, \bar{v}) \in dom(\theta_1) \\
\theta_2[\psi, \bar{v}] & (\psi, \bar{v}) \in dom(\theta_2)
\end{cases}
$$

We next define application of DSL rules to value maps. Let $rule = (ruleName, \psi_h, (\psi_1, \ldots, \psi_m))$ be a DSL rule. For $1 \leq i \leq m$, let $\bar{v}_i \in [\![\psi_i]\!]^n$ such that $\bar{v}_i = (v_{i,1}, \ldots, v_{i,n})$.

```
1:  function SynthProgram(DSL, CSR, φ)
2:      ψ := start symbol of DSL
3:      ê := root node of φ_O
4:      θ := SynthCSRStates(DSL, CSR, φ, ê, {ψ})
5:      return GetTopRankedProgram(φ, θ)
```

```
1:  function SynthCSRStates(DSL, CSR, φ, ê, ψ̃)
2:      θ_I := GetCSRTerminalValuesMap(CSR, φ)
3:      ψ̃_NT := non-terminal symbols of DSL
4:      Rules := rules of DSL
5:      let M map constituent-examples nodes to value maps
6:      for each child ê_c of ê do
7:          M[ê_c] := SynthCSRStates(DSL, CSR, ê_c, ψ̃_NT)
8:          θ_I := θ_I ∪ M[ê_c]
9:      θ_I := θ_I ∪ AggregatorRules(DSL, M)
10:     θ_I := θ_I ∪ ModifierRules(DSL, M)
11:     θ_R := GetCSRValuesMap(CSR, θ_I, φ, ê, ψ̃)
12:     ψ̃ := ψ̃ − Symbols(θ_R)
13:     while ψ̃ ≠ ∅ do
14:         θ_cur := θ_I ∪ θ_R
15:         θ_csr := ∅
16:         while θ_csr = ∅ do
17:             θ_cur := θ_cur ∪ ApplyRules(Rules, θ_cur)
18:             θ_csr := GetCSRValuesMap(CSR, θ_cur, φ, ê, ψ̃)
19:         θ_R := θ_R ∪ θ_csr
20:         ψ̃ := ψ̃ − Symbols(θ_R)
21:     return θ_R
```

Figure 4: Program synthesis algorithm

The lifting of rule application to value tuples is defined as $[\![rule]\!](\bar{v}_1, \ldots, \bar{v}_m) = \bar{v}$ such that $\bar{v} = (v_1, \ldots, v_n)$ and for $1 \leq k \leq n$ we have $v_k = [\![rule]\!](v_{1,k}, \ldots, v_{m,k})$. Rule application for value maps is defined as $[\![rule]\!](\theta_1, \ldots, \theta_m) = \theta$ such that $\theta[\psi, \bar{v}] = \tilde{P}$ if and only if there exist $\bar{v}_1, \ldots, \bar{v}_m$ such that $\theta[\psi_i, \bar{v}_i] = \tilde{P}_i$, $\bar{v} = [\![rule]\!](\bar{v}_1, \ldots, \bar{v}_m)$ and

$$\tilde{P} = \{P \mid P = ruleName(P_1, \ldots, P_m) \wedge P_i \in \tilde{P}_i\}$$

We write $[\![rule]\!](\theta) = [\![rule]\!](\theta_1, \ldots, \theta_m)$ if $\theta_i = \theta$ for all i. For a rule set $Rules$, we define

$$\mathsf{ApplyRules}(Rules, \theta) = \bigcup_{rule \in Rules} [\![rule]\!](\theta)$$

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 $\hat{e}$ from ϕ_O and a set of symbols $\tilde{\psi}$. It returns a value map which, for each symbol $\psi \in \tilde{\psi}$, contains programs of type ψ that satisfy the $CSR\langle\psi\rangle$ with respect to the examples node $\hat{e}$. 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 θ_I. 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 θ_I is first initialized with all the terminal components specified in the CSR terminal state sets. This is done by the GetCSRTerminalValuesMap function:

GetCSRTerminalValuesMap(CSR, ϕ)
  let n be the number of examples in ϕ
  return θ_r such that $\theta_r[\psi, \bar{v}] = v$ for all terminal symbols $\psi, v \in CSR\langle\psi\rangle$ and $\bar{v} = (v, \ldots, v)$ of size n.

Then, for each child of the node $\hat{e}$, there is a recursive call to generate components satisfying the child node and these components are added to θ_I (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 *aggregator* rules, which are binary recursive rules of the form $(ruleName, \psi, (\psi, \psi))$. 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 $ruleName(P_1, \ldots, P_n)$, e.g. tasks requiring a sequence of concatenations or disjunctions. For each aggregator rule $rule = (ruleName, \psi, (\psi, \psi))$ in the DSL and sub-sequence of child nodes $\hat{e}_i, \ldots \hat{e}_j$, the function AggregatorRules performs the rule application $[\![rule]\!](M[\hat{e}_i]|_\psi, \ldots, M[\hat{e}_j]|_\psi)$.

Similarly, the *modifier* rule pattern applies unary recursive rules of the form $(ruleName, \psi, (\psi))$. 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 $\hat{e}_c$, the function ModifierRules performs the rule application $[\![rule]\!](M[\hat{e}_c]|_\psi)$.

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

GetCSRValuesMap($CSR, \theta, \phi, \hat{e}, \tilde{\psi}$)
  return θ_r such that $\theta_r[\psi, \bar{v}] = \theta[\psi, \bar{v}]$ iff $\psi \in \tilde{\psi} \wedge CSR\langle\psi\rangle(\phi, \hat{e}, \bar{v})$ and undefined otherwise

The value map θ_R collects CSR states for all the required symbols. Using the initial components from θ_I, 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 ranking scheme our algorithm uses is based on a combination of the amount of CSR-satisfying

	FF	B1	B2	CPS
Number of timeouts	0	26	7	6
Number of incorrect results	46	15	6	0
Number of correct results	2	7	35	42
Average time (seconds)	< 0.5	12.35	8.99	9.97

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

components the program contains (which indicates the relevance of the program with respect to the given constituent examples and literal values), and the size of the program (favouring "simpler" programs based on the *Occam's Razor* approach of [Gulwani, 2011]). 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 (may include duplicate occurrences). 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

The evaluation of our approach is based on string manipulation tasks from online help forums for Excel and regular expressions [1]. These help forums illustrate the numerous difficulties users face in the string manipulation domain, as well as their need to express intent using both natural language and examples. We evaluated our compositional synthesis system on a set of 48 tasks taken from these forums. These tasks are covered by the wide range of constructs provided by DSL_s, including conditionals, loops, the various string transformation operators and complex regular expressions. Of the 48 tasks, 40 are not expressible in the DSLs of [Gulwani, 2011; Manshadi *et al.*, 2013] (require disjunctions, iterations, literals), and 20 are not expressible in the DSL of [Kushman and Barzilay, 2013] (require conditionals, loops, look-arounds and other string transformations).

For each task, we performed our evaluation using 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, our 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 was 1.53 (maximum 4 , minimum 0). The average execution time was 9.97 seconds, with 28 tasks completing in under 4 seconds. The programs synthesized by our system contained an average of 7 distinct DSL functions (minimum 4, maximum11), and had an average syntax tree size of 10.5 nodes (minimum 5, maximum 25).

We also compared our full system 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 on the same input-output examples given to our system). 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 advocated in [Gulwani, 2011]. In this case 35 programs were correctly synthesized, showing the benefits of compositionality not only in the tractability of search, but in the accuracy of ranking as well.

The timeouts on the 6 tasks can be explained by the exponential nature of the systematic search performed by the algorithm. In practice we expect significant optimizations to be obtained when we incorporate natural language understanding, based on training that the system can receive as tasks are performed. However, such NL-training by itself is inadequate and data-dependent e.g. [Kushman and Barzilay, 2013] report only 65% task coverage on their regular expression DSL which is much simpler than ours. We demonstrate 87% task-coverage without any training and with very few examples. Further gains are expected by incorporating NL-training, but in this work we have avoided such optimizations in order to evaluate the compositional approach independently of training data.

Discussion of sample forum tasks In Figure 6 we illustrate six of the help forum tasks used in our evaluation. For each task we show the relevant fragment of the original task description as given by the user in the help forum, the web address of the forum question, the examples given to our system and the program that was synthesized. Details of every task handled by our system can be found in [Raza *et al.*, 2015].

Task 6a illustrates how our system handles a string replacement task where the replacement is to be done on only part of the string that is to be matched. The task description may suggest the use of a conditional to check the existence of a 16 digit number, but this does not ensure that the string being replaced would be part of the one that is matched in the condition. Instead, after synthesizing the correct components $\mathsf{Interval}(\mathsf{NumChar}, 12)$ and $\mathsf{Interval}(\mathsf{NumChar}, 16)$, our system uses the lookahead operator to ensure correct substring replacement. This shows how the generated program may not always have a straightforward correspondence with the task description, as seen in other cases such as in Figure 1.

Task 6b illustrates a relatively bigger program using position expressions. Since the position expressions in the SubStr constructor are independent of one another, they can easily be used incorrectly in extraction tasks. For example, for Task 6b

[1] www.forums.devshed.com/regex-programming-147, www.stackoverflow.com, www.mrexcel.com

	Ex 1	Ex 2
Input	a5424180123456789c	c4015b1234567890123
Output	axxxxXXXXxxxx6789c	c4015b1234567890123
“a 16 digit number”	5424180123456789	
“the first 12 digits”	542418012345	

```
Replace(
  ConcatTok(
    LkAhead(Interval(NumChar, 16)),
    Interval(NumChar, 12)
  ),
  “xxxxXXXXxxxx”
)
```

(a) *If the cells contain **a 16 digit number** then -Replace **the first 12 digits** of each string with “xxxxXXXXxxxx”* (http://www.mrexcel.com/forum/excel-questions/712609-replace-numbers-string.html)

	Ex 1	Ex 2	Ex 3	Ex 4
Input	abc SN 12345 xyz	edf 242353 SN No. 421156 212311 mno	453abc11SN abc 12131232112	SN123
Output	12345	421156	12131232112	123
“some other text”	‘ ’	No.	abc	
“numbers”	12345	421156	12131232112	123

```
SubStr(
  Pos(ConcatTok(StrTok(“SN”),KleeneStar(Neg(NumChar))),KleenePlus(NumChar), 0),
  Pos(ConcatTok(StrTok(“SN”),ConcatTok(KleeneStar(Neg(NumChar)),KleenePlus(NumChar))),Optional(StrTok(“SN”)), 0)
)
```

(b) *extract any numbers afters “SN”. the numbers can be vary in digits. Also, at times there is **some other text** in between **numbers** and search word* (http://www.mrexcel.com/forum/excel-questions/763905-extract-numbers-after-specific-words.html)

	Ex 1	Ex 2
Input	Class (4) 1m5f Good	2 2 3 5M2Fxyz 3
Output	1m5f	5M2F
“a digit”	1	5
“the letter”	m	M
“a digit”	5	2
“the letter”	f	F

```
SubStr2(
  ConcatTok(
    NumChar,
    ConcatTok(
      DisjTok(CharTok(‘m’), CharTok(‘M’)),
      ConcatTok(NumChar, DisjTok(CharTok(‘f’), CharTok(‘F’)))
    )
  ),
  0
)
```

(c) ***a digit**, following by **the letter** “m” or “M”, followed by **a digit**, then followed by **the letter** “f” or “F”.* (http://stackoverflow.com/questions/18314280/java-regex-finding-a-characters-anywhere-in-a-string)

	Ex 1	Ex 2	Ex 3
Input	She had ringing in her ear.	mouth pain	Something else
Output	face	face	body

```
IfThenElse(
  Match(DisjTok(StrTok(“ear”), StrTok(“mouth”)), 0),
  ConstStr(“body”),
  ConstStr(“face”)
)
```

(d) *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”.* (http://www.mrexcel.com/forum/excel-questions/719190-if-cell-contains-certain-text-then-return.html)

	Ex 1	Ex 2	Ex 3	Ex 4
Input	1	1c	2	b
Output	1	1c	2	null
“any character”		c		

```
Filter(
  ConcatTok(DisjTok(CharTok(‘1’), CharTok(‘2’)), Optional(LowerChar))
)
```

(e) *The string must start with “1” or “2” (only once and mandatory) and then followed by **any character** between “a” to “z”(only once)* (http://stackoverflow.com/questions/16580566/regex-for-string-starts-with-numbers-and-followed-by-letters)

	Ex 1	Ex 2	Ex 3	Ex 4
Input	Asd	rZt35Hbeas	ra-	d2B
Output	Asd	rZt35Hbeas	null	null
“alphabet”	Asd	rZt		
“any alphanumeric”		35Hbeas		

```
Filter(
  ConcatTok(
    Interval(Union(LowerChar, UpperChar), 3),
    KleeneStar(Union(NumChar, Union(LowerChar, UpperChar)))
  )
)
```

(f) *first 3 character is **alphabet** (lower and upper both) then **any alphanumeric** if present* (http://forums.devshed.com/regex-programming-147/linux-regexp-machine-name-957229.html)

Figure 6: Sample forum tasks: the original task descriptions, the examples given to our system and the synthesized programs.

an incorrect expression for the second position may require only a number to match on the left. This may work in most cases where there is a single number in the string, but example 2 shows a case where this fails. As such, our system synthesizes the much longer position expression that ensures that the left matches the string "SN" followed by some sequence of characters, followed by a number. Also notice how the right-matching expression for the second position is bigger than required (an empty match would suffice), but that this does not affect the semantic correctness of the program.

Task 6c illustrates how our approach permits different semantic interpretations for the same natural language phrase used in different contexts. For example, the concept *"the letter"* appears twice in the task description, referring to two different abstractions. Based on the constituent examples, our system correctly infers the different disjunctions $\mathsf{DisjTok}(\mathsf{CharTok}(\text{`}m\text{'}), \mathsf{CharTok}(\text{`}M\text{'}))$ and $\mathsf{DisjTok}(\mathsf{CharTok}(\text{`}f\text{'}), \mathsf{CharTok}(\text{`}F\text{'}))$ for the two occurrences of this concept. This is due to the literal characters in the task specification and the requirement of least general character classes in the CSR relation. In contrast, for the concept *"a digit"* which also appears twice, the same abstraction NumChar is correctly inferred in both cases.

Task 6d illustrates the synthesis of a conditional program. No constituent examples are required for this task since the literal strings in the task description provide the correct CSR terminals and the aggregator rule optimization helps to efficiently generate the disjunctive condition.

Task 6e shows how with the help of examples our approach can achieve robustness with respect to inaccuracies or ambiguities that often occur in natural language descriptions of tasks. In this case the user states that the letter must appear "only once", but the four examples taken from his question indicate that the letter can occur *at most once or not all* (and this is indeed clarified by the user later on in the question). From these four clarifying examples our system is able to correctly apply the Optional modifier to the LowerChar token. Handling such ambiguities or inaccuracies may be difficult for purley language-based synthesis approaches such as [Kushman and Barzilay, 2013].

Task 6f illustrates another regular expression which uses disjunctive character classes. While our system generates the correct character class for the concept "alphanumeric" from the given constituent examples, this task also illustrates how compositional specifications with greater depths may be used. For instance, with a different CSR definition the system may not infer the correct composite character class for "alphanumeric". In this case, the user may elaborate on this concept and may describe it as "a number, lower case letter or upper case letter". Giving further constituent examples for each of these three concepts consititutes a further level in the compositional specification tree, and our system synthesizes the correct program given such a deeper specification.

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 indepedently 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 synthesis techniques 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 paradigm as we have demonstrated in this work.

The closely related area of *Programming-by-Demonstration* [Lau *et al.*, 2003b] 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 approach in the particular domain of string manipulation, supporting a very expressive domain language and evaluating with complex tasks from online help forums which are outside the scope of current state-of-the-art methods.

Given the domain-independent nature of our algorithm, in future work we plan to explore applications to other domains such as numerical algorithms [Lau *et al.*, 2003a]. Such explorations may also require further development of the language-based decomposition technique: in this work we have focussed on noun phrases as representing concepts for which examples may be given, but different decomposition approaches may be more suited to other possibly imperative (rather than functional) domains.

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

[Allen *et al.*, 2007] James F. Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary D. Swift, and William Taysom. Plow: A collaborative task learning agent. In *AAAI*, pages 1514–1519. AAAI Press, 2007.

[Chen and Mooney, 2011] David L. Chen and Raymond J. Mooney. Learning to interpret natural language navigation instructions from observations. In Wolfram Burgard and Dan Roth, editors, *AAAI*. AAAI Press, 2011.

[Clarke *et al.*, 2010] James Clarke, Dan Goldwasser, Ming-Wei Chang, and Dan Roth. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL-2010)*, pages 18–27, Uppsala, Sweden, 2010.

[Gulwani and Marron, 2014] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 803–814, New York, NY, USA, 2014. ACM.

[Gulwani *et al.*, 2012] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8), 2012.

[Gulwani, 2011] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. In *Principles of Programming Languages (POPL)*, pages 317–330, 2011.

[Katayama, 2007] Susumu Katayama. Systematic search for lambda expressions. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007.

[Klein and Manning, 2003] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 423–430, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[Kushman and Barzilay, 2013] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *HLT-NAACL*, pages 826–836. The Association for Computational Linguistics, 2013.

[Lau *et al.*, 2003a] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In John H. Gennari, Bruce W. Porter, and Yolanda Gil, editors, *K-CAP*, pages 36–43. ACM, 2003.

[Lau *et al.*, 2003b] Tessa A. Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by Demonstration Using Version Space Algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[Le and Gulwani, 2014] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, 2014.

[Liang *et al.*, 2011] Percy Liang, Michael I. Jordan, and Dan Klein. Learning dependency-based compositional semantics. *CoRR*, abs/1109.6841, 2011.

[Lieberman, 2001] Henry Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers, 2001.

[Manshadi *et al.*, 2013] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating programming by example and natural language programming. In Marie desJardins and Michael L. Littman, editors, *AAAI*. AAAI Press, 2013.

[Perelman *et al.*, 2014] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, 2014.

[Quirk *et al.*, 2012] Chris Quirk, Pallavi Choudhury, Jianfeng Gao, Hisami Suzuki, Kristina Toutanova, Michael Gamon, Wen tau Yih, Colin Cherry, and Lucy Vanderwende. Msr splat, a language analysis toolkit. In *HLT-NAACL*, pages 21–24. The Association for Computational Linguistics, 2012.

[Raza *et al.*, 2014] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Programming by example using least general generalizations. In *AAAI*, 2014.

[Raza *et al.*, 2015] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. Microsoft Research Technical Report MSR-TR-2015-33, 2015.

[Zettlemoyer and Collins, 2009] Luke S. Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In Keh-Yih Su, Jian Su, and Janyce Wiebe, editors, *ACL/IJCNLP*, pages 976–984. The Association for Computer Linguistics, 2009.