Programming by Example using Least General Generalizations

Mohammad Raza
Microsoft Research Cambridge
21 Station Road,
Cambridge, CB1 2FB, U.K.
a-moraza@microsoft.com

Sumit Gulwani
Microsoft Research Redmond
One Microsoft Way,
Redmond, WA 98052-6399, U.S.A.
sumitg@microsoft.com

Natasa Milic-Frayling
Microsoft Research Cambridge
21 Station Road,
Cambridge, CB1 2FB, U.K.
natasamf@microsoft.com

Abstract

Recent advances in Programming by Example (PBE) have supported new applications to text editing, but existing approaches are limited to simple text strings. In this paper we address transformations in richly formatted documents, using an approach based on the idea of *least general generalizations* from inductive inference, which avoids the scalability issues faced by state-of-the-art PBE methods. We describe a novel domain specific language (DSL) that expresses transformations over XML structures describing richly formatted content, and a synthesis algorithm that generates a minimal program with respect to a natural subsumption ordering in our DSL. We present experimental results on tasks collected from online help forums, showing an average of 4.17 examples required for task completion.

Introduction

The area of Programming by Example, or PBE (Lieberman 2001; Gulwani 2012), has recently been gaining renewed interest, especially in the domain of text editing (Gulwani 2011; Manshadi, Gildea, and Allen 2013; Liang, Jordan, and Klein 2010; Lau et al. 2003) and has also seen successful adoption in commercial end user applications such as the recent *Flash Fill* feature in Microsoft Excel (Gulwani 2011). These advances have, however, been limited to relatively simple editing scenarios on small unstructured text strings and cannot, for example, address structural transformations in richly formatted documents, e.g., word processing or presentation documents created by standard office productivity tools. Performing repetitive formatting operations in such documents is a common and demanding task, particularly for long documents or across large document collections, as we observed from many user requests in online help forums. Existing features such as styles or templates allow some degree of abstraction, but are limited in functionality and suffer from discoverability issues. On the other hand, macro programming languages are very powerful but above the skillset of most end users. We propose a PBE interaction that would naturally be suited to such tasks, allowing the user to provide some examples of the desired transformations using the standard WYSIWYG editing interface, from which the system can infer a general program to apply to whole documents.

An important challenge faced by PBE approaches in general is the delicate balance that must be achieved between the expressivity of the domain specific language (DSL) of transformations and the efficiency of the synthesis algorithm, because the more expressive the DSL, the more complex is the search space of possible programs, and the harder it is to effectively maintain and search within this space. For instance, recent state-of-the-art approaches use DAG based data structures to handle sophistication of the underlying transformation language (Gulwani 2011; Singh and Gulwani 2012b; Manshadi, Gildea, and Allen 2013; Singh and Gulwani 2012a). However, they only scale to relatively small unstructured strings (for example, Flash Fill only permits transformations on strings up to 256 characters) because of their complexity that is exponential in the number of examples and high degree polynomial in the size of each example. Hence, such approaches, even if they could express the underlying transformations, would not scale to our setting of transformations between large XML structures that describe richly formatted documents (most modern office suites work with XML-based file formats[1]).

In this paper we propose a PBE approach to structural format transformations which, in contrast to above-mentioned methods, avoids explicit generation of all consistent programs by incorporating the ranking strategy implicitly into the synthesis algorithm. After all, the end goal in any PBE task is to find a *single* satisfying program rather than the set of all possible satisfying programs.

Our approach is based on the notion of *least general generalization* from Plotkin's work on inductive inference (Plotkin 1970; 1971), which laid the foundations for the field of inductive logic programming. In this work the θ-*subsumption* relation was proposed as a tractable alternative to implication when inferring generalizations over first order formulae, and the syntactic anti-unification algorithm was described for inferring a least generalization with respect to subsumption. In our case, we are not working with logic formulae; our first contribution is a DSL that expresses transformations on ordered trees (including variable, function, and

[1]including Open Office, Microsoft Office, Apple iWork

loop expressions) and we show how programs in this DSL are naturally ordered by a similar subsumption relation. Our second contribution is a program synthesis algorithm which, given a set of input-output examples, generates a program in our DSL that satisfies all the examples. Using techniques similar to syntactic anti-unification, the algorithm generates a minimal program with respect to the subsumption ordering, without constructing a represention of all possible satisfying programs, which can be exponential in the size of examples.

Motivation

In a repetitive formatting task, the users usually have some intended selection criteria to specify items to change, which, in general, may be independent from the actual transformation they want to apply to these items, as we have observed in help forums. The selection criteria may, for example, be *all text boxes containing text with Arial font of size 12*, while the desired alteration may be *change every such text box to a table with a particular configuration*. Hence, the programs in our DSL must accurately characterise both the selection criteria for the applicable inputs as well as the desired outputs to ensure that only items relevant to the task are affected.

We consider an example of the above text box to table transformation task in Figure 1, where we use a simplified XML format to illustrate the state before and after the transformation in Figures 1(a) and 1(b) (attributes at nodes are shown inside boxes). Initially the text box contains three paragraphs (with text "P1", "P2" and "P3"), and satisfies the requirement of Arial font of size 12. In the output this is changed to a table with a certain height, border, font type and color, but the same text content and bold settings as the input. In this simplified format we only show some properties for illustration although, in practice, specifications contain many more elements, attributes, and substructures.

In Figures 1(c) and 1(d) we illustrate a program in our DSL that satisfies this example, specifying both the selection criteria and the transformation intended for this task. A program consists of two expressions describing the form of the applicable inputs and the outputs. The input expression in Figure 1(c) describes a tree that has a root element $textBox$ with a certain number of child subtrees that each has a root element $para$, with font Arial and size 12, and any text value, color or bold setting (expressed by variables X_1, X_2, X_3). The output expression in Figure 1(d) specifies a tree with root $table$ and *the same* number of child subtrees as the input (expressed by iteration variable I_1). Each subtree has root $tableRow$ and two children specifying row properties and a single cell. The text and bold values in the cell are the same as in the input (variables X_1 and X_3) while color, size, and font are constant.

While this program specifies precisely the selection criterion and the transformation that is intended for this task, there are many other possible programs in the DSL that can satisfy the example transformation in Figures 1(a) and 1(b). For example, instead of a loop the program may assert exactly 3 paragraphs in the input or, instead of a single cell in each row, there could be a loop expression to allow for multiple cells. Similarly for attribute values, the input expression may have specified variables over all the attributes, or specify the value "blue" for color rather than the variable X_2. Using all the examples provided for a given task, our program synthesis algorithm makes such choices by inferring a minimal generalization over the input and the output states and by mapping variables between the input and output states to generate a well-formed program satisfying all the examples.

Domain Specific Language

In this section we define the model of XML specifications that we use in this paper and the syntax and semantics of the domain specific language (DSL) used to express transformations on these structures. We model XML specifications as ordered trees in which every node has an associated element and a map from attributes to values.

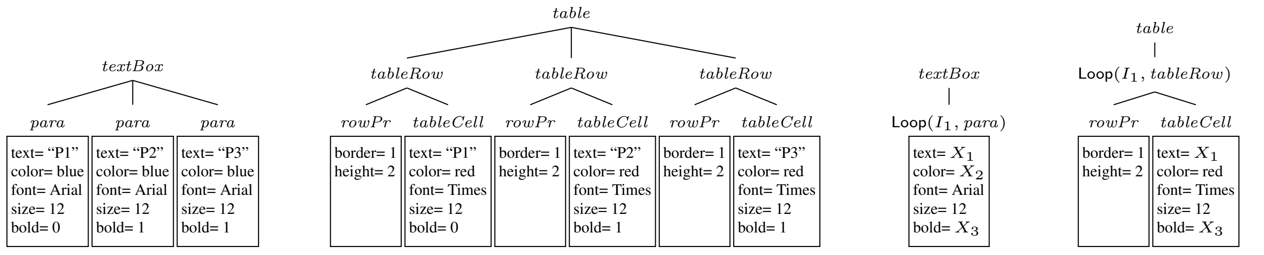
Definition 1 (Tree model) *Given fixed sets of elements* Elm*, attributes* Att *and literal values* Lit*, we define trees* $t \in T$ *by the following grammar:*

$$t := 0 \mid r \mid t \cdot t \qquad r := (e, m)[t]$$

where 0 is the empty tree, r *is a rooted tree and* $t \cdot t$ *is a parallel composition of trees as ordered siblings. A rooted tree* r *is of the form* $(e, m)[t]$ *with root node* (e, m) *and subtree* t*. A node* (e, m) *consists of an element* $e \in \mathsf{Elm}$ *and a partial map from attributes to literal values* $m : \mathsf{Att} \rightharpoonup \mathsf{Lit}$*. We impose the structural congruences that parallel composition is associative, i.e.,* $(t_1 \cdot t_2) \cdot t_3 = t_1 \cdot (t_2 \cdot t_3)$*, and has identity* 0*, i.e.,* $t \cdot 0 = 0 \cdot t = t$*.*

We next define the DSL for transforming XML trees which is agnostic of any particular XML schemas and hence generically applicable. The syntax of the DSL is given in Figure 2, which we first discuss before giving the formal semantics in the next section. A program $\mathsf{Prog}(\tau_1, \tau_2)$ consists of a pair of tree expressions τ_1, τ_2, where τ_1 describes the applicable input trees to the program and τ_2 describes the output tree. For example, Figures 1(c) and 1(d) illustrate the input and output tree expressions for the text to table program. A tree expression is a parallel composition of atomic expressions, where an atomic expression is either a rooted tree expression or a loop expression.

A rooted tree expression $(e, \phi)[\tau]$ describes a rooted tree that has root element e, attribute map described by ϕ and subtree described by τ. An attribute map expression ϕ may use literals, variables or function expressions to express the values that a particular attribute may take. For example, in Figure 1(d), there is a rooted subtree with root element $tableCell$ at which the 'text' and 'bold' attributes have variable values while 'color', 'font' and 'size' attributes have literal values. Function expressions may be used to describe transformations of attribute values from input to output. For example, in Figure 1(d) we could have 'text' = $f(X_1)$ to indicate that the text value at that node changes according to some text transformation function f (e.g., syntactic transformations such as name or date format changes (Gulwani 2011)). On numeric-valued attributes we may have, for example, 'size' = X in the input and 'size' = X + 3 in the out-



textBox
para
para
para
text= “P1” color= blue font= Arial size= 12 bold= 0
text= “P2” color= blue font= Arial size= 12 bold= 1
text= “P3” color= blue font= Arial size= 12 bold= 1
table
tableRow
rowPr
tableCell
border= 1 height= 2
text= “P1” color= red font= Times size= 12 bold= 0
text= “P2” color= red font= Times size= 12 bold= 1
text= “P3” color= red font= Times size= 12 bold= 1
Loop(I1, para)
text= X1 color= X2 font= Arial size= 12 bold= X3
Loop(I1, tableRow)
text= X1 color= red font= Times size= 12 bold= X3

(a) Example input state (b) Example output state (c) Program input (d) Program output

Figure 1: Example input and output states and satisfying program for the text box to table transformation task

put. In general, we assume a fixed set of functions between literal values that are available in our DSL.

Loop expressions allow for structural variations in the trees by expressing an arbitrary number of sibling trees in a parallel composition. An expression $\mathsf{Loop}(I, \rho)$ describes any number of sibling trees, each described by rooted tree expression ρ. The loop iterator I is used to relate loop expressions between input and output tree expressions. For example, in Figure 1 there are loop expressions in the program input and output that are related by the iterator I_1. When applying the program to an input such as in Figure 1(a), the iterator I_1 is used to determine the number of loop iterations when instantiating the output (3 in this case) as well as the valuations of the variables for each iteration (X_1, X_2 and X_3).

Note that the language does not permit nested loop expressions at the same level of parallel composition (e.g. $(I_1, (I_2, \rho))$ but loop expressions can be nested inside subtrees (e.g. (I_1, ρ) where ρ contains I_2). For example, if every paragraph in Figure 1(a) were fragmented into a number of runs and each run should appear in a separate column in the output table, that may be expressed by a program in our DSL by using a nested loop expression to represent varying number of table cells in each table row.

We define $\mathsf{Var}(\tau)$, $\mathsf{Iter}(\tau)$ and $\mathsf{FExp}(\tau)$ for the set of variables, iterators, and function expressions that appear in tree expression τ respectively. These sets are similarly defined for programs, atomic and rooted tree expressions, and attribute map expressions. We define the set of root elements in τ as $\mathsf{Root}(\tau) =$

$$\begin{cases} \emptyset & \tau = 0 \\ \{e\} & \tau = (e, \phi)[\tau'] \text{ or } \tau = \mathsf{Loop}(I, (e, \phi)[\tau']) \\ \bigcup_{i \in 1..n} \mathsf{Root}(\alpha_i) & \tau = \alpha_1 \cdot \ldots \cdot \alpha_n \end{cases}$$

The well-formedness constraint (†) in Figure 2 ensures unambiguous matching of expressions with concrete trees, as well as tractable inference in the presence of loops. Note that every concrete tree is a valid tree expression, since any $t \in T$ is an expression τ with $\mathsf{Var}(\tau) = \mathsf{Iter}(\tau) = \mathsf{FExp}(\tau) = \emptyset$.

Semantics

When a program $\mathsf{Prog}(\tau_1, \tau_2)$ is applied to a concrete tree t, a valuation of variables and iterators is inferred for the input tree expression τ_1 to match t. This valuation is then applied to the output tree expression τ_2 to instantiate the output tree that is the result of the program. If no valuation can be found to convert the output to a concrete tree then the program is not applicable on the given input tree. Formally, we have

$$[\![\mathsf{Prog}(\tau_1, \tau_2)]\!]t = \begin{cases} t' & \sigma = \mathsf{Match}(\tau_1, t) \wedge t' = \mathsf{Apply}(\tau_2, \sigma) \wedge t' \in T \\ \bot & \text{otherwise} \end{cases}$$

where the valuation is represented by a subsititution σ, and the Match and Apply functions are used to generate σ from the input and apply it to the output. A substitution maps variables to literal values and iterators to a sequence of substitutions in which each substitution represents an iteration of the loop and contains the valuation of variables to apply in that iteration.

Definition 2 (Substitution) *A substitution $\sigma \in \mathsf{Subs}$ maps variables to literals and iterators to a sequence of substitutions. Thus $\sigma : (\mathsf{Var} \rightharpoonup \mathsf{Lit}) \cup (\mathsf{Iter} \rightharpoonup \mathsf{Subs}^n)$ where $n \in \mathbb{N}$. A substitution is a partial map and we write $\sigma(X) = \bot$ or $\sigma(I) = \bot$ when σ is undefined on X or I.*

We say σ and σ' are compatible, written $\sigma \# \sigma'$, iff $\sigma(X) = \sigma'(X)$ on all X in which both are defined and if for any I we have $\sigma(I) = (\sigma_1, ..., \sigma_n)$ and $\sigma'(I) = (\sigma'_1, ..., \sigma'_m)$ then $n = m$ and $\sigma_i \# \sigma'_i$ for all i.

$$\begin{array}{rcl}
\mathsf{Program}\ P & := & \mathsf{Prog}(\tau_1, \tau_2) \\
\mathsf{TreeExp}\ \tau & := & 0 \mid \alpha \mid \tau \cdot \tau \quad \text{with constraint (†)} \\
\mathsf{AtomicTreeExp}\ \alpha & := & \rho \mid \mathsf{Loop}(I, \rho) \quad \text{where } I \in \mathsf{Iter} \\
\mathsf{RootedTreeExp}\ \rho & := & (e, \phi)[\tau] \quad \text{where } e \in \mathsf{Elm} \\
\mathsf{AttMapExp}\ \phi & : & \mathsf{Att} \rightharpoonup \mathsf{Val} \\
\mathsf{Val} & := & \mathsf{Lit} \cup \mathsf{Var} \cup \mathsf{FExp} \\
\mathsf{FExp} & := & f(X) \text{ where } f : \mathsf{Lit} \to \mathsf{Lit} \wedge X \in \mathtt{Var} \\
\mathsf{Var} & := & X, X_1, X_2, \ldots \\
\mathsf{Iter} & := & I, I_1, I_2, \ldots
\end{array}$$

(†) if $\tau = \alpha_1 \cdot \ldots \cdot \alpha_n$ then τ is well-formed if
- $\forall j < k < \ell.\ \mathsf{Root}(\alpha_j) = \mathsf{Root}(\alpha_\ell) \Rightarrow \mathsf{Root}(\alpha_j) = \mathsf{Root}(\alpha_k)$
- if $\alpha_i = \mathsf{Loop}(I, \rho)$ then $\forall j \neq i.\ \mathsf{Root}(\alpha_i) \neq \mathsf{Root}(\alpha_j)$

Figure 2: Syntax of domain specific language

$$\mathsf{Match}(\tau,t)=\begin{cases}\emptyset & \tau=0 \text{ and } t=0\\ \mathsf{Match}(\phi,m)\cup\mathsf{Match}(\tau_1,t_1)\cup\mathsf{Match}(\tau_2,t_2) & \tau=(e,\phi)[\tau_1]\cdot\tau_2 \text{ and } t=(e,m)[t_1]\cdot t_2\\ \mathsf{Match}(\tau',t')\cup[I\mapsto(\mathsf{Match}(\rho,r_1),\ldots,\mathsf{Match}(\rho,r_n))] & \tau=\mathsf{Loop}(I,\rho)\cdot\tau' \text{ and } t=r_1\cdot\ldots\cdot r_n\cdot t' \text{ where}\\ & \forall i.\mathsf{Root}(r_i)=\mathsf{Root}(\rho) \text{ and } \mathsf{Root}(\rho)\not\subseteq\mathsf{Root}(\tau')\\ \bot & \text{otherwise}\end{cases}$$

$$\mathsf{Apply}(\tau,\sigma)=\begin{cases}0 & \tau=0\\ (e,\mathsf{Apply}(\phi,\sigma))[\mathsf{Apply}(\tau',\sigma)] & \tau=(e,\phi)[\tau']\\ \mathsf{Apply}(\rho,\sigma_1)\cdot\ldots\cdot\mathsf{Apply}(\rho,\sigma_n) & \tau=\mathsf{Loop}(I,\rho)\wedge\\ & \sigma(I)=(\sigma_1,\ldots,\sigma_n)\\ \mathsf{Loop}(I,\rho) & \tau=\mathsf{Loop}(I,\rho)\wedge\\ & \sigma(I)=\bot\\ \mathsf{Apply}(\alpha,\sigma)\cdot\mathsf{Apply}(\tau',\sigma) & \tau=\alpha\cdot\tau'\end{cases}$$

$$\mathsf{Match}(\phi,m)=\begin{cases}\bigcup_{\phi(a)\in\mathtt{Var}}[\phi(a)\mapsto m(a)] & \forall a.\phi(a)\notin\mathtt{Var}\Leftrightarrow\phi(a)=m(a)\\ \bot & \text{otherwise}\end{cases}$$

$\mathsf{Apply}(\phi,\sigma)=\phi'$ such that $dom(\phi')=dom(\phi)$ and

$$\phi'(a)=\begin{cases}\ell & \phi(a)=X\wedge\sigma(X)=\ell\\ f(\ell) & \phi(a)=f(X)\wedge\sigma(X)=\ell\\ \phi(a) & \text{otherwise}\end{cases}$$

Figure 3: Match and Apply functions used in the semantics of programs

The union of substitutions $\sigma\cup\sigma'$ is undefined if σ and σ' are not compatible, and is otherwise defined as σ_u as follows. For variable X we have $\sigma_u(X)=\sigma(X)$ or $\sigma_u(X)=\sigma'(X)$ if either are defined, and $\sigma_u(X)=\bot$ otherwise. For iterator I we have $\sigma_u(I)=$

$$\begin{cases}\sigma(I) & \sigma(I)\neq\bot\wedge\sigma'(I)=\bot\\ \sigma'(I) & \sigma'(I)\neq\bot\wedge\sigma(I)=\bot\\ (\sigma_1\cup\sigma_1',\ldots,\sigma_n\cup\sigma_n') & \sigma(I)=(\sigma_1,\ldots,\sigma_n)\wedge\\ & \sigma'(I)=(\sigma_1',\ldots,\sigma_n')\\ \bot & \textit{otherwise}\end{cases}$$

The definitions of the Match and Apply functions are given in Figure 3 where they are defined recursively over the inductive structure of tree and attribute map expressions. The $\mathsf{Match}(\tau,t)$ function returns a substitution that matches expression τ with concrete tree t. It builds such a substitution by taking the union of substitutions from separate substructures; the substitution is undefined if any of these unions are undefined. The $\mathsf{Apply}(\tau,\sigma)$ function instantiates variables and loops in expression τ using the substitution σ, and any variables or iterators in τ that are not in σ remain uninstantiated.

Programs in our DSL have a natural generality ordering given by a subsumption relation with respect to substitutions, which is analagous to the subsumption relation between first order formulae given in (Plotkin 1970). The program synthesis algorithm we describe in the next section infers programs that are minimal with respect to this ordering.

Definition 3 (Subsumption) *For programs $P=(\tau_1,\tau_2)$ and $P'=(\tau_1',\tau_2')$ we say that P subsumes P', written $P'\leq P$, iff there exists a substitution σ such that* $\mathsf{Apply}(\tau_1,\sigma)=\tau_1'$ *and* $\mathsf{Apply}(\tau_2,\sigma)=\tau_2'$.

Synthesis Algorithm

In this section we describe the program synthesis algorithm which, given a set of input-output examples of trees, returns a program in our DSL that satisfies all the examples. In summary, the algorithm proceeds by constructing least general tree expressions over all the input trees and all the output trees, and then relates the variables in these two expressions to construct a valid satisfying program. The algorithm is defined in Figure 4, where the following definitions are used.

For a tree t and an element e, we define $\mathsf{FirstRoot}(t,e)$ iff $t=r\cdot t'$ and $\mathsf{Root}(r)=\{e\}$. We define a *scope* $s\in\mathsf{Scopes}$ as a sequence of iterators to represent a nested loop context, so that

$$s:=\emptyset\mid I\mid I*I$$

where $*$ is sequence concatenation. We also define

$$\Psi_{\mathsf{Var}}:\mathsf{Scopes}\times\mathsf{Val}^n\rightarrow\mathsf{Var}$$

to be an injective map that assigns a distinct variable for a given scope and sequence of n values, and

$$\Psi_{\mathsf{Iter}}:\mathsf{Scopes}\times\mathbb{N}^n\rightarrow\mathsf{Iter}$$

to be an injective map that assigns a distinct iterator for a given scope and sequence of n numbers. The maps Ψ_{Var} and Ψ_{Iter} are used to generate unique variables and iterators when constructing expressions, which is a technique adapted from the first order syntactical anti-unification algorithm of (Plotkin 1970; 1971). In our case, since the generalizations are over tree expressions that may include variables and nested loop expressions, we define the mappings to be conditional on scopes in order to prevent name clashes in different nested loop contexts.

To infer transformations between literal values, we assume a given black box function

$$\mathsf{InferLiteralFunction}:\mathcal{P}(\mathsf{Lit})\rightarrow(\mathsf{Lit}\rightarrow\mathsf{Lit})$$

which infers a function that satisfies the given input-output examples of literals or returns $\bot$ if no such function can be generated. Our implementation supports linear numeric transformations and syntactic string transformations using (Gulwani 2011).

```
 1: function $\mathsf{InferProgram}((t_1, t'_1), \ldots, (t_n, t'_n))$
 2:     $\tau_1 := \mathsf{InferTreeExp}(\emptyset, (t_1, \ldots, t_n))$
 3:     $\tau_2 := \mathsf{InferTreeExp}(\emptyset, (t'_1, \ldots, t'_n))$
 4:     for all $X \in (\mathsf{Var}(\tau_2) - \mathsf{Var}(\tau_1))$ do
 5:         for all $X' \in \mathsf{Var}(\tau_1)$ in the same scope as $X$ do
 6:             $f := \mathsf{InferLiteralFunction}(\mathsf{GetLiterals}(X', X))$
 7:             if $f \neq \bot$ then
 8:                 $\tau_2 := \tau_2[X \backslash f(X')]$
 9:     if $\mathtt{Var}(\tau_2) \not\subseteq \mathtt{Var}(\tau_1) \vee \mathsf{Iter}(\tau_2) \not\subseteq \mathsf{Iter}(\tau_1)$ then
10:         FAIL
11:     return $\mathsf{Prog}(\tau_1, \tau_2)$
```

```
 1: function $\mathsf{InferTreeExp}(s, (t_1, \ldots, t_n))$
 2:     if $\forall i.\ t_i = 0$ then
 3:         return $0$
 4:     let $e$ be such that $\exists j.\ \mathsf{FirstRoot}(t_j, e)$ and
          $\forall i.\ e \in \mathsf{Root}(t_i) \Rightarrow \mathsf{FirstRoot}(t_i, e)$
 5:     if no such $e$ exists then
 6:         FAIL
 7:     for all $i = 1 \ldots n$ do
 8:         let $t_i = r_{i,1} \cdot \ldots \cdot r_{i,M_i} \cdot t'_i$ where
              $e \notin \mathsf{Root}(t'_i)$ and $\forall j.\ \mathsf{Root}(r_{i,j}) = \{e\}$
 9:     $\tau := \mathsf{InferTreeExp}(s, (t'_1, \ldots, t'_n))$
10:     if $M_1 = \ldots = M_n$ then
11:         for all $j = 1 \ldots M_1$ do
12:             $\rho_j := \mathsf{InferRootExp}(s, (r_{1,j}, \ldots, r_{n,j}))$
13:         return $\rho_1 \cdot \ldots \cdot \rho_{M_1} \cdot \tau$
14:     $I := \Psi_{\mathsf{Iter}}(s, M_1, \ldots, M_n)$
15:     $\rho := \mathsf{InferRootExp}(s * I, (r_{1,1}, \ldots, r_{1,M_1}, r_{2,1}, \ldots, r_{n,M_n}))$
16:     return $\mathsf{Loop}(I, \rho) \cdot \tau$
```

```
 1: function $\mathsf{InferRootExp}(s, (r_1, \ldots, r_n))$
 2:     if $\forall i.\ r_i = (e, m_i)[t_i]$ then
 3:         $\phi := \mathsf{InferAttMap}(s, (m_1, \ldots, m_n))$
 4:         $\tau := \mathsf{InferTreeExp}(s, (t_1, \ldots, t_n))$
 5:         return $(e, \phi)[\tau]$
 6:     else FAIL
```

```
 1: function $\mathsf{InferAttMap}(s, (m_1, \ldots, m_n))$
 2:     if $\exists i, j.\ dom(m_i) \neq dom(m_j)$ then FAIL
 3:     let $\phi$ be a new attribute map
 4:     for all $a \in dom(m_1)$ do
 5:         if $\forall i.\ m_i(a) = v$ then
 6:             $\phi(a) := v$
 7:         else
 8:             $\phi(a) := \Psi_{\mathsf{Var}}(s, m_1(a), \ldots, m_n(a))$
 9:     return $\phi$
```

```
 1: function $\mathsf{GetLiterals}(X_1, X_2)$
 2:     let $(s, v_1, \ldots, v_n) = \Psi^{-1}_{\mathsf{Var}}(X_1)$
 3:     let $(s', v'_1, \ldots, v'_m) = \Psi^{-1}_{\mathsf{Var}}(X_2)$
 4:     if $n \neq m \vee s \neq s'$ then return $\bot$
 5:     $R := \emptyset$
 6:     for all $i = 1 \ldots n$ do
 7:         if $v_i, v'_i \in \mathsf{Var}$ then
 8:             $R := R \cup \mathsf{GetLiterals}(v_i, v'_i)$
 9:         else if $v_i, v'_i \in \mathsf{Lit}$ then
10:             $R := R \cup \{(v_i, v'_i)\}$
11:         else return $\bot$
12:     return $R$
```

Figure 4: Program synthesis algorithm

We next describe each of the functions in Figure 4. The main function is InferProgram, which returns a program from the given set of input-output pairs of trees. It starts by constructing tree expressions τ_1 and τ_2 over all the input trees and all the output trees (lines 1 and 2). It then attempts to replace every variable in τ_2 not appearing in τ_1 by a function of an input variable (lines 4-7), using InferLiteralFunction. The $\mathsf{GetLiterals}(X', X)$ function returns a set of pairs of literals that represent instantiations of the variables X' and X, as determined by the Ψ_{Var} mapping. A program constructed from τ_1 and τ_2 is returned at the end if τ_2 contains only variables and iterators appearing in τ_1.

InferTreeExp takes a scope and trees $t_1, \ldots, t_n$, and returns a tree expression that generalizes over the given trees (the scope represents the loop context and is empty in the first call). If all trees are empty then the empty tree is simply returned (line 3). At line 4, the algorithm chooses the next root element to generalize over, which is selected as an element that occurs as the first root in all the expressions if it occurs at all (if no such element exists e.g. e_1, e_2 and e_2, e_1, then failure occurs as no unifying expressions can be found in line with the well-formedness constraint (†) in Figure 2). The root element's number of occurrences in each tree t_i is given by M_i and t'_i are the rest of the sibling trees (line 8). At line 9, a recursive call is made to find the generalization τ over the rest of the sibling trees.

If the root element has the same number of occurrences in all the given trees, then loops need not be inferred on this element. Hence at line 12, a rooted tree expression ρ_j is inferred for each occurrence of the root element over all the examples and these are concatenated together with the result of the recursive call to give the final result. However, if the root element does not have the same number of occurrences among all examples, then a loop expression must be inferred for this element. To infer a loop, the algorithm uses the Ψ_{Iter} map to generate a new iterator based on the occurrences of the root element in each of the examples. The loop expression is then generated by inferring a rooted tree expression over all occurrences of the root element in all examples, under the scope of the new iterator (line 15).

InferRootExp infers a rooted tree expression given a scope and sequence of rooted trees. All given trees must have the same root element and the result is constructed as a combination of the generalizations inferred on the attribute maps and the subtrees recursively. InferAttMap takes a scope and attribute maps, and returns a generalizing attribute map expression that introduces variables over attributes that do not have a consistent value across all the given attribute maps.

Theorem 1 (Soundness and minimality) *If we have that* InferLiteralFunction *is sound and we have* $\mathsf{InferProgram}((t_1, t'_1), \ldots, (t_n, t'_n)) = P$ *then*

- $\forall i.\ [\![P]\!] t_i = t'_i$
- $\forall P'.\ (\forall i.\ [\![P']\!] t_i = t'_i) \wedge P' \leq P \Rightarrow P \leq P'$

The proof follows by induction, by checking that for the generalization created at every step, a satisfying minimal substitution exists for every given example. For instance, in the base case of InferAttMap, for each example m_i there exists a σ_i such that $\mathsf{Apply}(\phi, \sigma_i) = m_i$, where for every generated variable $X = \Psi_{\mathsf{Var}}(s, v_1, ..., v_n)$ in ϕ we have $\sigma_i(X) = v_i$.

Implementation and Extensions

We have implemented an add-in for Microsoft PowerPoint 2013, called *FlashFormat*, which automates repetitive editing tasks in richly formatted presentation documents based on examples that the user can provide using the standard PowerPoint editing interface. These edits are reflected in the underlying XML specification of the document which, in this case, is the Office Open XML file format (OOXML) (International Organization for Standardization 2008). The system is based on the DSL and the algorithm described above, and incorporates the following extensions to the core algorithm to optimize for certain cases that arise in practice.

Disjunctions. Some tasks may be expressed by a set of programs in our DSL if a single program cannot express the entire task. For instance, if a task involves different transformations on different inputs, a set of programs can be used to represent a case statement describing different selection conditions and corresponding outputs, e.g., if the user gives examples of text boxes changing colour and arrows becoming thicker, then this may be expressed as a disjunction of two programs describing each transformation separately. Although such disjunctive tasks could in theory be accomplished by the user through a succession of independent subtasks, our system applies the following optimization: if for a given set of examples a single program that covers all the examples cannot be found, then the system searches for a set of programs over a partition of the example set.

Domain specific information. A number of attributes and elements in the PowerPoint OOXML format relate to non-visual or meta properties of objects (e.g., object IDs, language settings) or minor visual properties (e.g., autofit options, space before/after). Our system is designed to avoid sensitivity of inferred programs to specific settings for a fixed set of such properties and always assigns variables rather than literal values for such properties when inferring generalizations.

Experimental Evaluation

We performed an evaluation of the system using examples from online help forums (http://www.msofficeforums.com, http://www.vbaexpress.com/forum). Forums serve a wide range of topics (e.g., help with VBA syntax, PowerPoint interface/features) and so we manually searched for questions about repetitive formatting tasks. In these requests the users describe repetitive tasks that they would like to automate and an expert usually provides a code in the macro programming language VBA that fulfills the task. We chose 18 such questions from PowerPoint forums and 2 questions from Word forums concerning tables that could also be applicable to editing in PowerPoint.

Figure 5 shows the results of our evaluation on the 20 questions (T1 - T20) in which we applied our system to automate the specified task from examples. The tasks cover a wide range of formatting features in PowerPoint documents, including shapes, images, charts, tables, bullets, margins,

Task	R1	R2	R3	Avg
T1	2	2	2	2
T2	4	4	3	3.7
T3	4	5	5	4.7
T4	13	11	11	11.7
T5	3	3	2	2.7
T6	2	4	2	2.7
T7	4	2	3	3
T8	8	7	8	7.7
T9	3	3	3	3
T10	11	10	11	10.7

Task	R1	R2	R3	Avg
T11	2	4	2	2.7
T12	4	6	7	5.7
T13	2	2	2	2
T14	2	3	3	2.7
T15	2	2	2	2
T16	6	5	5	5.3
T17	3	4	3	3.3
T18	2	2	3	2.3
T19	2	3	3	2.7
T20	3	3	3	3

Figure 5: Number of examples required in 20 forum tasks

fonts, borders, size, positioning, scaling, colors, styles, and indents. In a few cases the task also involved requests other than formatting (e.g., importing images and then formatting), in which case we only addressed the formatting component. Most tasks had specific selection criteria on where to apply changes, e.g., *change only text of a certain font and size* (T11), *change only circle shapes that are red in color* (T15), *change the indentation of any text that has a certain font but only when it appears inside a table* (T19). Five of the tasks had no selection criteria (changes were required everywhere).

Since actual documents associated with the tasks were not available in the forum questions, we chose them either from a collection of files from the internet or, if the task described a particular document structure, then we created a sample document based on that description (e.g., presentation contains a chart on each slide). For each task we conducted three runs (R1, R2, R3), in which a different set of examples was given to the system. The number of examples required in all the runs are shown in Figure 5. Overall, an average of 4.17 examples were required for task completion (maximum 13 and minimum 2), with execution times normally under a second and not exceeding 5 seconds for any execution on any task. For 15 of the 20 tasks it was possible to generate a single (non-disjunctive) program in our DSL to complete the task.

Related Work

A number of Programming by Example techniques have been developed recently for automating repetitive data manipulation tasks, including syntactic and semantic string transformations (Gulwani 2011; Singh and Gulwani 2012a; Liang, Jordan, and Klein 2010; Menon et al. 2013), number transformations (Singh and Gulwani 2012b), and spreadsheet table layout transformations (Harris and Gulwani 2011). Programming by demonstration (PBD) is also a closely related technique (Lau et al. 2003; Witten and Mo 1993; Miller and Myers 2001) where transformations are inferred from the trace of actions that the user performs rather than the input and output states, as in PBE. The problem that we address and the method of inferring transformations in structured documents with XML representations are beyond the scope of all of this previous work. A recent work on integrating PBE with natural language programming (Manshadi, Gildea, and Allen 2013) shows that for syntactic string transformations fewer examples are required in the presence of natural language specifications. Thus, it would be interesting to explore ways to combine this method with our least general generalization approach.

A number of machine learning methods have been applied to learning XML document transformations in the context of information integration and extraction (Gilleron et al. 2006; Chidlovskii and Fuselier 2005; Tsochantaridis et al. 2005). In particular, (Gilleron et al. 2006) show how Conditional Random Fields (CRF) can be applied to achieve relatively high precision in node labelling with around 20 examples. However, the set of editing tasks that can be expressed through labelling of a predefined XML trees is limited. In general, in the interactive PBE setting the aim is to have a very small number of examples (usually less than 10), which is hard to achieve by statistical modelling methods. Our DSL has been designed specifically for this setting: it supports explicit specification of the selection criteria in input expressions to target the highly selective nature of the transformations we consider. This design is also in contrast to approaches using existing XML transformation languages such as XSLT (Stube 2004). Indeed, output tree expressions in such languages are explicitly defined but constraints on input structure would be more challenging to infer.

The technique of the *least general generalization* that we use is based on Plotkin's work on inductive inference (Plotkin 1970; 1971), which laid the foundations of inductive logic programming. A range of methods have since been explored in the inference of logic programs from examples (Muggleton and Feng 1990). We use Plotkin's ideas in a PBE setting where we formulate least generalizations in a novel DSL for expressing XML transformations. While our work lies outside of logic programming, interesting related work is on *feature terms* (At-Kaci and Podelski 1993; Carpenter 1992), a generalization of first order terms that are used in formalizing object oriented declarative languages. While set-valued features in feature terms bear a similarity to our iteration variables, the notion of subsumption in the former case is different. Indeed, it requires a 1-1 matching between elements in set-valued features while, in our case, the iteration variables allow a varying number of elements in ordered sequences. As described in (Ontanon and Meseguer 2012), set-valued features also lead to efficiency challenges when checking subsumption between feature terms while, in our ordered setting, subsumption can be checked in time that is linear in tree size.

Conclusion

We have described a Programming by Example approach which extends the applicability of the paradigm to the domain of structural transformations in richly formatted documents. This is based on the notion of least general generalization which provides an implicit ranking strategy and avoids the scalability issues in state of the art PBE systems. Our DSL and algorithm have been designed with a natural subsumption ordering on programs with respect to which we can find minimal programs efficiently, while also delivering expressivity sufficient to handle target transformations with support for variables, functions, and loop expressions in the language. Experiments on forum examples demonstrate the feasibility of the approach and its applicability to interactive editing of complex documents.

Apart from the evaluation of our method on forum examples presented here, we are also conducting user studies and exploring alternative user interfaces to observe how users respond to a system based on least generalizations and how they can effectively give examples to complete repetitive tasks. We also plan to explore the application of our approach to other structured transformation domains such as program transformations in software development environments (Meng, Kim, and McKinley 2011). This may require extending the DSL features, e.g., loops using more sophisticated boolean conditions. Increased expressivity usually means that more examples are needed in PBE approaches, and it would therefore be interesting to explore techniques such as (Manshadi, Gildea, and Allen 2013) to integrate natural language input with our approach in order to target more complex domains.

Acknowledgments

We would like to thank the anonymous reviewers for their very helpful feedback that has improved this paper.

References

At-Kaci, H., and Podelski, A. 1993. Towards a Meaning of LIFE. *J. Log. Program.* 16(3):195–234. preliminary version: PLILP 1991: 255-274.

Carpenter, B. 1992. *The Logic of Typed Feature Structures*. Cambridge: Cambridge University Press.

Chidlovskii, B., and Fuselier, J. 2005. A Probabilistic Learning Method for XML Annotation of Documents. In Kaelbling, L. P., and Saffiotti, A., eds., *IJCAI*, 1016–1021. Professional Book Center.

Gilleron, R.; Jousse, F.; Tellier, I.; and Tommasi, M. 2006. XML Document Transformation with Conditional Random Fields. In Fuhr, N.; Lalmas, M.; and Trotman, A., eds., *INEX*, volume 4518 of *Lecture Notes in Computer Science*, 525–539. Springer.

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

Gulwani, S. 2012. Synthesis from Examples: Interaction Models and Algorithms. In *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 8–14.

Harris, W. R., and Gulwani, S. 2011. Spreadsheet Table Transformations from Examples. In Hall, M. W., and Padua, D. A., eds., *Programming Language Design and Implementation (PLDI)*, 317–328. ACM.

International Organization for Standardization. 2008. Office Open XML File Formats. ISO/IEC 29500-1:2008.

Lau, T. A.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53(1-2):111–156.

Liang, P.; Jordan, M. I.; and Klein, D. 2010. Learning programs: A hierarchical bayesian approach. In Frnkranz, J., and Joachims, T., eds., *International Conference on Machine Learning*, 639–646. Omnipress.

Lieberman, H., ed. 2001. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers.

Manshadi, M. H.; Gildea, D.; and Allen, J. F. 2013. Integrating programming by example and natural language programming. In desJardins, M., and Littman, M. L., eds., *AAAI*. AAAI Press.

Meng, N.; Kim, M.; and McKinley, K. S. 2011. Systematic Editing: Generating Program Transformations from an Example. In Hall, M. W., and Padua, D. A., eds., *PLDI*, 329–342. ACM.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B. W.; and Kalai, A. 2013. A Machine Learning Framework for Programming by Example. In *ICML (1)*, volume 28 of *JMLR Proceedings*, 187–195. JMLR.org.

Miller, R. C., and Myers, B. A. 2001. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX Annual Technical Conference*, 161–174.

Muggleton, S., and Feng, C. 1990. Efficient Induction of Logic Programs. In *First Conference on Algorithmic Learning Theory*, 368–381.

Ontanon, S., and Meseguer, P. 2012. Efficient Operations in Feature Terms Using Constraint Programming. In Muggleton, S.; Tamaddoni-Nezhad, A.; and Lisi, F. A., eds., *ILP*, volume 7207 of *Lecture Notes in Computer Science*, 270–285. Springer.

Plotkin, G. D. 1970. A Note on Inductive Generalization. *Machine Intelligence* 5:153–163.

Plotkin, G. D. 1971. A Further Note on Inductive Generalization. *Machine Intelligence* 6:153–163.

Singh, R., and Gulwani, S. 2012a. Learning Semantic String Transformations from Examples. *PVLDB* 5(8):740–751.

Singh, R., and Gulwani, S. 2012b. Synthesizing Number Transformations from Input-Output Examples. In Madhusudan, P., and Seshia, S. A., eds., *Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, 634–651. Springer.

Stube, B. 2004. *Automatic Generation of XSLT by Simultaneous Editing*. Massachusetts Institute of Technology (PhD Thesis).

Tsochantaridis, I.; Joachims, T.; Hofmann, T.; and Altun, Y. 2005. Large Margin Methods for Structured and Interdependent Output Variables. *Journal of Machine Learning Research* 6:1453–1484.

Witten, I. H., and Mo, D. 1993. TELS: learning text editing tasks from examples. In *Watch what I do: programming by demonstration*, 183–203.