An Iterative Approach to Synthesize Data Transformation Programs*

Bo Wu
Computer Science Department
University of Southern California
Los Angeles, California
bowu@isi.edu

Craig A. Knoblock
Information Science Institute
University of Southern California
Los Angeles, California
knoblock@isi.edu

Abstract

Programming-by-Example approaches allow users to transform data by simply entering the target data. However, current methods do not scale well to complicated examples, where there are many examples or the examples are long. In this paper, we present an approach that exploits the fact that users iteratively provide examples. It reuses the previous subprograms to improve the efficiency in generating new programs. We evaluated the approach with a variety of transformation scenarios. The results show that the approach significantly reduces the time used to generate the transformation programs, especially in complicated scenarios.

1 Introduction

The development of programming-by-example approaches (PBE) allows users to perform data transformation without coding [Lieberman, 2001]. The user can directly enter the data in the target format and the system learns from these input-output examples to generate a transformation program that is consistent with the examples.

Examples often have many interpretations. Users generally provide multiple examples in an iterative way to clarify their intentions. For example, in Table 1, a user wants to extract the year, maker, model, location, and price from car sale posts. The left column shows the titles of the car sale posts and the right shows the target values. The user can directly enter the target data (2000 Ford Expedition los angeles $4900) for the first entry as an example. The system synthesizes a program and applies the program to the rest of the data. The user checks the transformed results and provides examples for any incorrect results to refine the program until she determines the results are correct.

PBE systems typically generate an entirely new program as users provide new examples. Their time complexity is exponential in the number and a high polynomial in the length of examples [Raza *et al.*, 2014]. This prevents these PBE systems from being applied to real-world scenarios that require many or long examples to clarify a user's intention.

*This research is based upon work supported in part by the National Science Foundation under Grant No. 1117913.

We observe that the programs generated from previous examples are usually close to the correct ones. A large portion of the programs remain the same as a user provides additional examples to refine the program. Reusing the correct subprograms can greatly reduce the computational costs.

We present an approach to adapt programs with additional examples. To adapt the program, first, the approach needs to identify the incorrect subprograms. Since a transformation program often has multiple subprograms, it is essential to correctly identify the incorrect subprograms to avoid missing them or redundantly generating the correct subprograms. Second, the approach needs to be able to generate correct subprograms to replace the incorrect ones.

To address the above challenges, we have two insights. First, we noticed that PBE approaches typically generate traces. Traces are the computational steps executed by a program to yield an output from a particular input [Kitzelmann and Schmid, 2006]. A *trace* defines how the output string is constructed from a specific set of substrings from the input string. The PBE approaches then generalize over these traces to produce the programs that are consistent with all examples [Summers, 1977; Kitzelmann and Schmid, 2006; Gulwani, 2011; Lau *et al.*, 2003; Harris and Gulwani, 2011; Singh and Gulwani, 2012b; 2012a]. As these traces encode the required input and output pairs for each subprogram, they can be leveraged to detect the buggy subprograms. Second, the correctness of a program generated by PBE is only determined by whether it can output the correct results specified by the traces. Thus, if a program can return the expected results on the examples, the program is considered to be correct even though the program may fail on future unseen examples.

Our approach can deterministically identify incorrect subprograms and adapt them to additional examples. When the user provides a new example, our approach applies the previously learned program on this new example. It records the outputs of all the subprograms and compares them against the expected outputs shown in the trace of the example. As the number of incorrect subprograms can be different from the number of outputs in the trace, our approach precisely maps subprograms to their corresponding expected outputs to identify the incorrect subprograms, whose execution results differ from the expected ones. As the transformation program has subprograms, our approach searches for the incorrect subprograms until no more incorrect subprograms is available. The

Table 1: Data transformation scenario

Input Data	Target Data
2000 Ford Expedition 11k runs great los angeles $4900 (los angeles)	2000 Ford Expedition los angeles $4900
1998 Honda Civic 12k miles s. Auto. - $3800 (Arcadia)	1998 Honda Civic Arcadia $3800
2008 Mitsubishi Galant ES $7500 (Sylmar CA) pic	2008 Mitsubishi Galant Sylmar CA $7500
...	...

approach then generates new correct subprograms to replace incorrect subprograms. The new subprograms are consistent with both previous and new examples.

To sum up, our approach makes the following contributions:

- iteratively generating programs from examples,
- deterministically identifying incorrect subprograms and refining them,
- enabling the PBE approach to scale to much more complicated examples.

2 Previous Work

Our approach builds on the state-of-the-art PBE system described in [Gulwani, 2011]. His approach defines a Domain-Specific Language (DSL). This language supports a restricted, but expressive form of regular expressions, which includes conditionals and loops. The approach synthesizes transformation programs from this language using examples.

To generate a transformation program, Gulwani's approach follows these steps: First, it creates all traces of the examples. It segments the outputs into all possible combinations. From the original input, it then generates these segments, independently of each other, by either copying substrings from original values or inserting constant strings. One trace example in Figure 1 shows that the target value is decomposed into 2 segments (solid brackets). These 2 segments are copied from the original input. A transformation program is composed of a set of subprograms that produce the segments and the corresponding positions in the input string. A trace shows the expected outputs of each of these subprograms. For example, one segment program will output "2000 Ford Expedition " and its start position program will output '0' and the end position program will output '20' etc. Another trace from the example split the output into 3 segments and copies "los angeles" from a different place (dashed bracket). It succinctly stores all these traces using a directed acyclic graph where nodes correspond to the positions and edges correspond to the segments.

Second, it derives hypothesis spaces from the traces. A *hypothesis space* defines the set of possible programs that can transform the input to the output. A program defines the locations of the strings in the input using either constant positions, constant tokens, or token types. For example, the hypothesis space used to describe the first segment "2000 Ford Expedition " has two alternatives to describe the program (1) as a constant string or (2) using start and end positions. The start position can be described as an absolute value ('0') or using the context of the position. For instance, the left context of the start position can be described as the beginning of the input and the right context of the start position is "2000" or a number type. When there are multiple examples, the approach merges the hypothesis spaces from each example to find the common hypothesis space for all examples. For example, the right context of the first record for the start position is "2000" and the right context of the second record for the same position is "1998". Thus, the right context that is common for the two examples is a number type rather than a specific number.

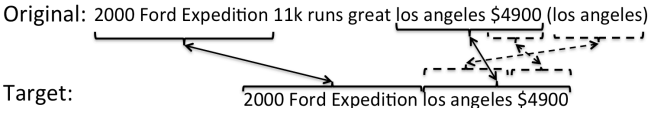
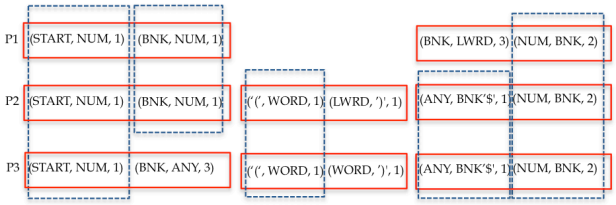


Figure 1: Segmentations of the first example

```
Transform(val)
    pos1=val.indexOf(START, NUM, 1)
    pos2=val.indexOf(BNK, NUM, 1)
    pos3=val.indexOf(BNK, LWRD, 3)
    pos4=val.indexOf(NUM, BNK, 2)
    output = val.substr(pos1, pos2) + val.substr(pos3, pos4)
    return output
```

(a) Program learned from the first example



(b) Programs refined by adding the second and third row

Figure 2: Program changes as more examples are added

Finally, it generates the programs from the hypothesis space based on their simplicity. The approach always generates simpler programs first based on the partial order defined on the transformation language. For example, it generates the program with fewer segments earlier. If a single branch transformation program cannot cover all the examples, the algorithm partitions the examples and generates a partition transformation program for each partition individually. A conditional expression can be learned to distinguish these partitions. This approach also supports loop expressions by detecting whether contiguous segments can be merged.

Figure 2(a) shows the program learned using the first record as an example. The program is basically a concatenation of several segment programs. A segment program aims to generate a substring such as "$val.substr(pos_1, pos_2)$" or a constant string. The two parameters (pos_1, pos_2) refer to the

start and end position programs of the substring in the original value. Each position program as pos_1=*val.indexOf(START, NUM, 1)* consists of three components (left context, right context, occurrence). The left and right contexts specify the surrounding tokens of the target location. An *occurrence (n)* refers to the n-th appearance of a position with the context. A token is a sequence of characters. The approach uses the following tokens: NUM ([0-9]+), LWRD([a-z]+), UWRD([A-Z]), WORD([a-zA-Z]+), BNK(whitespace), ANY(match any token) and punctuation such as a comma, a hyphen, etc. The UWRD represents a single uppercase letter, LWRD represents a sequence of lowercase letters, WORD represents a sequence of any letters and NUM refers to a sequence of digits. START and END tokens indicate the start and end positions of the input. Hence, pos_2 refers to the first occurrence of a position whose left is a blank token and right context is a NUM.

The transformation program is refined as the user provides new examples as shown in Figure 2(b). $P1$ is the synthesized program from the first row of Table 1, $P2$ is the program when the first two rows are used as examples and the $P3$ is the returned program from the first three rows. The solid rectangles represent segment programs and each segment program has two position expressions in parentheses. The dashed rectangles show the position programs that stay the same as the program changes. We can see that a large portion of the new program does not change. From $P1$ to $P2$, 3 out 4 position programs stay the same. From $P2$ to $P3$, 4 out of 6 are the same as before.

3 Iteratively Learning Programs by Example

Our approach adapts programs with new examples by identifying incorrect subprograms and replacing them with refined programs. We first introduce some notations. Let $P = [p_1, p_2, ..., p_n]$ represent the transformation program. Every $p_i = const|(p_i^s, p_i^e)$ corresponds to a segment program. It can be either a constant string or extracting substring from the input. p_i^s and p_i^e are the start and end position programs. A position program identifies a position with certain context in the input. $P_{[k,l]}$ refers to a subsequence of programs between index k and l ($1 \leqslant k \leqslant l \leqslant n$). Let $T^{'} = [t_{p_1}, t_{p_2}, ..., t_{p_n}]$ represent the output of the program P on the new example. The t_{p_i} is the execution result of the corresponding subprograms p_i. The $t_{p_i^s}$ and $t_{p_i^e}$ are the corresponding execution results of p_i^s and p_i^e. Let $T = [t_1, t_2, ..., t_m]$ represent the trace created from the new example. t_i is a segment trace, which defines how a substring in the output is produced from the input. If the substring is copied from input value, t_i^s and t_i^e refer to the start and end positions of the segment in the input. Let $H = \{H_1 = [h_{11}], H_2 = [h_{21}, h_{22}]...\}$ represent the hypothesis space used to generate programs of different numbers of segments. H_i represents the space that has i segments. For example, H_2 defines the set of possible programs that create the output string using two segments. A *segment hypothesis space* contains all possible programs for generating a segment. We use h_{ij} to represent all the jth segment programs in the programs with i segments. $H_{i[r,t]}$ represents the subsequence of segment spaces between index r and t of H_i. Similarly, h_{ij}^s and h_{ij}^e correspond to the start and end position spaces of h_{ij} containing start and end position programs.

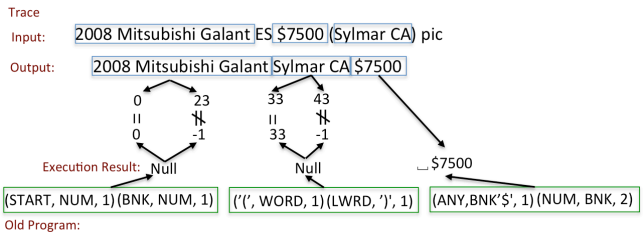


Figure 3: P and T have same number of segments

To adapt the transformation program on the new example, our approach creates traces from the new example. It then iterates over these traces and utilizes these traces to generate a list of patches. Each **patch** contains 3 elements: (1) a subsequence of incorrect programs, (2) their expected traces and (3) their corresponding hypothesis spaces. The approach then uses these traces in the patch to update the corresponding hypothesis spaces and generates correct subprograms to replace the incorrect subprograms. The approach stops when it either successfully generates a transformation program that is consistent with all examples or it exhausts all the traces.

To refine a program using a trace with m segments, the approach selects the hypothesis space H_m from H, which contains all the candidate programs with m segments. Depending on the number of segments (n) in the current program (P), it handles two cases separately: (1) $n = m$ and (2) $n \neq m$.

3.1 P and T have the same number of segments

The number of the execution results of the segment programs is the same as the number of segment traces. The approach directly compares the t_{p_i} with the t_i (line 3). If any execution result differs from the trace, there is an incorrect segment program. The algorithm adds this program (p_i) with the corresponding trace (t_i) and hypothesis space (h_{mi}) for further refinement.

For example, Figure 3 shows a program learned using the first two records in Table 1. The new example is the third record. In the figure, we represent each segment program using a rectangle with its start and end position programs in the parentheses. The "Null" in the execution results represents that the segment program cannot generate an output. The "-1" means the program cannot find a position matching the pattern specified in the position program. The first segment program in Figure 3 cannot generate the correct substring (2008 Mitsubishi Galant) as specified in the trace. The algorithm adds the program, its trace and corresponding segment hypothesis space into the patches for refinement (line 4). As the third segment program's output is correct, it doesn't need to be refined.

3.2 P and T have a different number of segments

Since the number of segment programs is different from the number of segment traces, the algorithm cannot directly map the segment programs to the segment traces with the same indices. However, it can find the mapping between segment

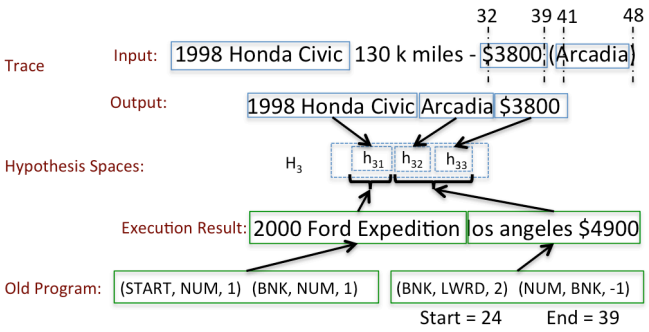


Figure 4: P and T have different number of segments

programs and segment hypothesis spaces. Meanwhile, the segment hypothesis spaces are mapped to segment traces with the same indices. The algorithm can then align the segment traces and segment programs since they are mapped to the same sequence of segment hypothesis spaces.

The algorithm maps subsequences of segment programs to subsequences of segment hypothesis spaces (line 5). To identify the mapping, it first obtains the output ($O_{P_{[k,l]}}$) by evaluating the subsequence of programs ($P_{[j,k]}$) on old examples (O). It then identifies the hypothesis space ($H_{m[i,j]}$). The part of old examples with the output $O_{H_{m[i,j]}}$ used to derived $H_{m[i,j]}$ should contain the same string as $O_{P[k,l]}$. Thus, the space ($H_{m[i,j]}$) contains sequences of subprograms that can generate the same output as $P_{[k,l]}$ but these sequences of subprograms can represent different segmentations.

Figure 4 shows the outputs of two segment programs. The p_2 generates "los angeles $4900". H_3 contains a different segmentation where the last two segment hypothesis spaces (h_{32} and h_{33}) are derived from the traces with output "los angeles " and "$4900" separately. Thus, the second segment program is mapped to the second and third segment hypothesis spaces in H_3. The algorithm can then identify the aligned segment traces using the same indices as the segment hypothesis spaces. It then compares whether the subprograms can generate the results as specified in the traces or whether the lengths of the two sequences match to decide whether it should add this sequence of programs for further adaptation (line 7).

The algorithm can further map the position programs with the traces. When a sequence of segment programs are mapped to a sequence of segment traces, the approach compares the output of the first start and the last end position programs with the first start and the last end position traces to exclude the correct subprograms that can generate the expected results (line 8 and 9). For example, the last segment program in Figure 4 is mapped to two segment traces. The end position of the third segment trace is 39, which is the same as the output of p_2^e on the new example (39). Thus, the current end position program is correct and can be reused. In the case that P and T have the same number of segments, there is only one segment in $T_{[i,j]}$ and $P_{[k,l]}$. The start position program of the first segment ($t_{p_1^s}$) outputs "0", which is the same as the start position in the trace(t_1^s) in Figure 3. Therefore, the algorithm excludes this position program from adaptation.

3.3 Adapting Incorrect Programs

As the algorithm has identified the patches consisting of incorrect subprograms, the expected traces and the corresponding hypothesis spaces, it first uses the traces to update these spaces. The algorithm first creates a basic hypothesis space using the traces and merges this space with the identified hypothesis spaces to generate the updated spaces using the same method described in the previous work section. It then generates the correct subprograms from the updated spaces that is consistent with expected traces. Finally, it replaces the incorrect subprograms with correct subprograms and returns the new program (line 10).

Algorithm 1: Program Adaptation

Input: P program, H hypothesis space, T trace of the new example, O old examples
Output: P_{new}
n = size(P), m = size(T), patches = []
1 H_m = findHypothesisSpaceByLength(H, m)
2 **if** $n = m$ **then**
  for i = *[1, m]* **do**
3 **if** $t_i \neq t_{p_i}$ **then**
4 patches.add (($[p_i]$, $[t_i]$, $[h_{mi}]$))
  end
  end
else
5 seqmap= $\{([k,l] : [i,j]) \mid O_{P_{[k,l]}} \in O_{H_{m[i,j]}}\}$
6 **for** *{ [k, l] : [i, j] } in seqmap* **do**
7 **if** $(j-i) \neq (l-k) \vee T_{[i,j]} \neq T_{P_{[k,l]}}$ **then**
  patches.add(($P_{[k,l]}$, $T_{[i,j]}$, $H_{m[i,j]}$))
  end
  end
end
for *($P_{[k,l]}$, $T_{[i,j]}$, $H_{m[i,j]}$) in patches* **do**
8 **if** $t_i^s = t_{p_k^s}^s$ **then** modify patch to remove p_k^s
9 **if** $t_j^e = t_{p_l^e}$ **then** modify patch to remove p_l^e
end
10 P_{new} = apply(patches, P)
return P_{new}

3.4 Soundness and Completeness

Our approach can always adapt the transformation program using the new example to generate a correct program, if there exists a correct transformation program.

Proof 1 *The approach is sound as it only returns the program that is consistent with examples. To prove the completeness, suppose $\exists P^*$ consistent with $O \cup N$. O refers to the previous examples and N is the new example. This implies $\exists trace^*$ $trace^*$ is the trace of the correct program (P^*) on the new example (N).* **First, the algorithm can identify all incorrect subprograms** *w as it only excludes the correct subprograms that generate expected outputs specified by $trace^*$.* **Second, the identified space H_i contains the correct subprograms.** *As the w is consistent with O, to replace w, the correct subprograms (r) should also generate the same output as w on*

previous examples O. As the recovered space H_i contains all the alternative programs that can generate the same outputs as w, the space contains r. Lastly, as the approach uses a brute force search in the space to identify the correct subprograms r as described in [Gulwani, 2011], it can identify the correct subprograms r. Therefore, the algorithm can generate P^* by replacing w with r.

3.5 Performance Optimizations

There can be multiple traces for one input-output pair. To more efficiently adapt the programs, our approach filters traces and then sorts the remaining traces to reuse most of the previous subprograms.

The trace (T) should always have at least the number of segments as the number of segments in the program (P). Because the approach generates simpler programs with fewer segments first from all the programs that are consistent with examples, all the programs with fewer segments have been tested and failed to transform the examples correctly. Therefore, the approach only uses the traces with a larger or equal number of segments to refine the program.

We aim to make the fewest changes to the program to make it consistent with the new example. The approach sorts the traces in descending order based on their resemblance to the T'. The approach iterates over the traces in the sorted list to adapt the program. To sort the traces, the approach creates a set s_1 that contains the outputs of the position expressions. It then creates a set s_2 that contains all the positions in the trace. The approach then sorts the traces based on the score $(size(s_1 \cap s_2)+1)/(size(s_2)+1)$. The high score indicates a large overlap between s_1 and s_2. It in turn means a close resemblance between the program and the trace.

4 Evaluation

We implemented our iterative programming-by-example approach (IPBE) as part of Karma [Knoblock and Szekely, 2015][1] . We conducted an evaluation on both real-world and synthetic datasets.

4.1 Dataset

Our real-world data consists of two parts. First, we collected the 17 scenarios from [Lin *et al.*, 2014] (referred to as D1). Each scenario contains 5 records. We also gathered 30 scenarios used in [Wu and Knoblock, 2014] (referred to as D2). The average number of records in each scenario is about 350.

Second, we created a synthetic dataset by combining multiple scenarios. The synthetic scenarios are to transform records with multiple fields at the same time. We show three example input and output records in Table 2. They have 7 columns in the output records. To transform one record from the input to the output, the approach learns a sequence of transformations, such as extract the first name, extract the last name, etc., and combines them in one transformation program. By changing the number of columns (1- 10) in the output records, we can control the complexity of the scenario. Each scenario has about 100 records.

[1]Data and code are available at http://bit.ly/1GtZ4Gc. The code is also available as the data transformation tool of Karma (http://www.isi.edu/integration/karma).

Table 2: Synthetic scenario for generating the first 7 columns

	Name		Year		Dimension			...
input	Cook Peter		1905 - 1998. (T.V)		22 x 16 1/8 x 5 1/4 inches			...
	Clancy Tom		1858 - 1937		5/8 x 40 x 21 3/8 inches			...
	Hicks Dan		1743 - 1812		6 15/16 x 5 1/16 x 8 inches			...
	First	Last	Birth	Death	1st	2nd	3rd	...
output	Cook	Peter	1905	1998	22	16 1/8	5 1/4	...
	Clancy	Tom	1858	1937	5/8	40	21 3/8	...
	Hicks	Dan	1743	1812	6 15/16	5 1/16	8	...

4.2 Experiment Setup

We performed the experiments on a laptop with 8G RAM and 2.66GHz CPU. We compared IPBE with two other approaches: (1) Gulwani's approach [Gulwani, 2011] and (2) $Metagol_{DF}$ [Lin *et al.*, 2014]. We used our own implementation of Gulwani's approach rather than Flashfill in Excel 2013, as a large portion of scenarios in the test data cannot be transformed by Flashfill, and there is no easy way to instrument Excel to accurately measure the program generation time. For $Metagol_{DF}$, we obtained the code from the authors and ran the code on our machine to obtain the results on D1. We compared the three methods in terms of the time (in seconds) to generate a program that is consistent with the examples.

4.3 Real-World Scenario Results

The results on real world scenarios are shown in Table 3. We calculated the average program generation time (in seconds) for each scenario. To calculate the time for IPBE and Gulwani's approach, we recorded the program generation time for all the iterations until all the records were transformed correctly. We then averaged the program generation time across all iterations and refer to this average time as the generation time. For $Metagol_{DF}$, we used the experiment setting in [Lin *et al.*, 2014] and averaged the program generation time for the same scenario.

The Min is the shortest time among the set of generation time for all scenarios. The Max, Avg and Median are also calculated for the same set generation time. A 0 in the results means the time is smaller than one millisecond. As we cannot easily change $Metagol_{DF}$ to run it on D2, we only ran our approach and Gulwani's approach on D2.

We can see that IPBE outperforms the other two approaches as shown in Table 3. Comparing IPBE with Gulwani's approach, we can see that reusing previous subprograms can improve the system efficiency. $Metagol_{DF}$ treats each character as a token, which enables it to do transformation on the character level. However, it also significantly increases the search space, which causes the system to spend more time to induce the programs.

4.4 Synthetic Scenarios Results

To study how Gulwani's approach and IPBE scale on complex scenarios, we created 10 synthetic scenarios with the number of columns in the output ranging from 1 to 10. We gradually increased the number of columns so that the approaches had to learn more complicated programs. We ran

Table 3: Results of real-world scenarios

		Min	Max	Avg	Median
D1	IPBE	0	5	0.34	0
	Gulwani's approach	0	8	0.59	0
	Metagol	0	213.93	55.1	0.14
D2	IPBE	0	1.28	0.20	0
	Gulwani's approach	0	17.95	4.02	0.33
	Metagol	~	~	~	~

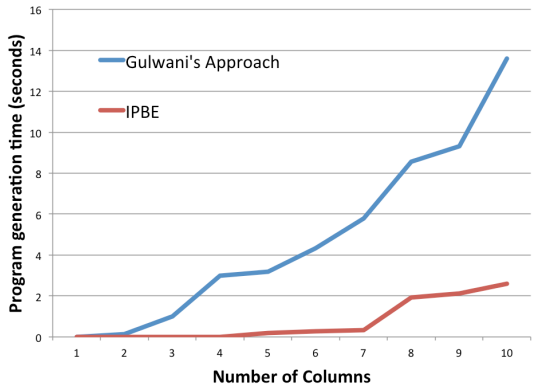


Figure 5: Synthesizing time rises as column number increases

the two approaches and provided examples until they learned the programs that can transform all records correctly and then measured the average time used to generate a program.

The time in Figure 5 used by the two approaches increases as the number of columns increased. However, IPBE scales much better compared to Gulwani's approach when more columns are added. The time saving comes from the fact that IPBE can identify the correct subprograms and only refine the incorrect subprograms. The cost of generating a new program is related to the portion of the program that requires updating rather than the actual size of the program. A program with more subprograms can usually reuse more subprograms so that the time saving will be more evident.

5 Related Work

PBE approaches have been extensively studied for the past decades. Early work [Kushmerick, 1997] [Hsu and Dung, 1998] [Muslea *et al.*, 1999] in wrapper induction learns extraction rules from user labels to extract target fields from documents. [Lau *et al.*, 2003] proposed an approach to derive text-editing programs from a sequence of user edit operations. However, these approaches typically require separate labels for each field or labels for each step.

The pioneering work in program induction [Summers, 1977] can induce Lisp programs with one recursive function from the traces of input-output pairs. [Kitzelmann and Schmid, 2006] extended this approach to induce a set of recursive equations with more than one recursive call. Recently, researchers developed approaches to induce programs for data transformation [Gulwani, 2011; Harris and Gulwani, 2011; Singh and Gulwani, 2012b; Raza *et al.*, 2014; Manshadi *et al.*, 2013]. These approaches introduce the Domain Specific Language (DSL) to support a set of predefined transformations. They learn the hypothesis space based on the DSL and then generate programs. However, these approaches do not utilize the correct subprograms generated previously.

More recently, several approaches show that reusing the previous subprograms is promising. [Perelman *et al.*, 2014] focuses on developing an approach to synthesize programs for various domains given the DSL. Their approach can reuse previous subprograms. It maintains two sets: (1) one set called contexts containing the programs with part of its subprograms deleted to create holes and (2) the other set containing all the subprograms from previously generated programs. Through implanting the subprograms into the holes in the contexts, it can create new programs. [Lin *et al.*, 2014] uses the meta-interpretive learning framework [Muggleton and Lin, 2013] to learn domain specific bias. By reusing the predicates generated from other tasks or previous iterations, their approach can use fewer examples and generate programs more efficiently. Our work is orthogonal to these works. These works focus on maintaining a library of previous generated subprograms and reusing these programs when encountering new examples. As the number of subprograms in the library keeps increasing, searching in this library for the right subprograms can require more time. Our approach takes advantage of traces to deterministically identify, refine incorrect subprograms and reuse correct subprograms.

For automatic program bug repair, [Shapiro, 1991] developed an approach to deterministically adapt programs with new evidence using resolution tree backtracking. Recently, approaches using generic programming to generate fairly complicated software patches have been applied to automatic bug fixes [Weimer *et al.*, 2010; Goues *et al.*, 2012]. These approaches often require either an oracle to test whether certain parts of the program are correct or require a large number of test cases to locate the problem. Our approach is different from these approaches as it automatically creates the expected outputs for the subprograms using the given examples.

6 Conclusion and Future Work

This paper presents a program adaptation approach for programming-by-example systems. Our approach identifies previous incorrect subprograms and replaces them with correct subprograms. The experiment results show that our approach significantly reduces the time to generate transformation programs and is more scalable on complicated scenarios. It enables the PBE approach to induce programs in real time, which greatly enhances the usability of such systems in a broad range of scenarios.

In the future, we will improve the work in two aspects. First, our approach relies on the traces that have the input-output pairs for all the subprograms. We will extend our approach to domains where we can only obtain partial traces that contain only input-output pairs for a subset of subprograms. Second, having more informative examples in early iterations can reduce the required number of examples. We plan to develop an approach to help users provide more informative examples first.

References

[Goues *et al.*, 2012] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng*, 2012.

[Gulwani, 2011] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[Harris and Gulwani, 2011] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *SIGPLAN*, 2011.

[Hsu and Dung, 1998] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.*, 1998.

[Kitzelmann and Schmid, 2006] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 2006.

[Knoblock and Szekely, 2015] Craig A. Knoblock and Pedro Szekely. Exploiting semantics for big data integration. *AI Magazine*, 2015.

[Kushmerick, 1997] Nicholas Kushmerick. *Wrapper Induction for Information Extraction*. PhD thesis, 1997.

[Lau *et al.*, 2003] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 2003.

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

[Lin *et al.*, 2014] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI*, 2014.

[Manshadi *et al.*, 2013] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating programming by example and natural language programming. In *AAAI*, 2013.

[Muggleton and Lin, 2013] Stephen Muggleton and Dianhuan Lin. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. In *IJCAI*, 2013.

[Muslea *et al.*, 1999] Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In *AGENTS*, 1999.

[Perelman *et al.*, 2014] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

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

[Shapiro, 1991] Ehud Y. Shapiro. Inductive inference of theories from facts. In *Computational Logic - Essays in Honor of Alan Robinson*, 1991.

[Singh and Gulwani, 2012a] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 2012.

[Singh and Gulwani, 2012b] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, 2012.

[Summers, 1977] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 1977.

[Weimer *et al.*, 2010] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 2010.

[Wu and Knoblock, 2014] Bo Wu and Craig A. Knoblock. Iteratively learning conditional statements in transforming data by example. In *Proceedings of the First Workshop on Data Integration and Application at the 2014 IEEE International Conference on Data Mining*. IEEE, 2014.