Learning Data Transformation Rules through Examples: Preliminary Results*

Bo Wu
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA
bowu@isi.edu

Pedro Szekely
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA
pszekely@isi.edu

Craig A.Knoblock
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA
knoblock@isi.edu

ABSTRACT

A key problem in many data integration tasks is that data is often in the wrong format and needs to be converted into a different format. This can be a very time consuming and tedious task. In this paper we propose an approach that can learn data transformations automatically from examples. Our approach not only identifies the transformations that are consistent with all examples, but also recommends the transformations that most likely transform the rest of unseen data correctly. The experimental results show that in six transformation scenarios our approach produces good results.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; I.2.2 [**Artifical Intelligence**]: Automatic Programming—*Program synthesis*

General Terms

Design

Keywords

data transformation, program synthesis

1. INTRODUCTION

Currently, many applications rely on data gathered from multiple data sources with different formats. To utilize the data, users often need to normalize it, transforming each type of data into a single format. Figure 1 shows two examples. Suppose the user has a data set with addresses where some have the city name at the beginning and some have the city name at the end. The user would like to normalize the data so that all the city names are at the beginning. To do so, the user must transform those addresses that have the city name at the end, moving the city name to the beginning, moving the comma between city name and the street address. The figure also shows an example with phone numbers where the user wants to insert parenthesis around the first three digits and delete the hyphen. In general, transformations involve a sequence of operations that reorder, insert and delete tokens. The objective of this work is to enable users to easily construct transformation programs that they can apply to entire data sets.

One common approach to construct such data transformation programs is to use regular expressions. For example, the transformations for the example shown in Figure 1 can be written in Perl as follows:

- s/([^,])+,([^,])+/$2,$1
- s/(\d){3}-(.+)/($1)$2

Writing scripts such as these is difficult for most users. In order to ease the burden of manually writing transformation programs, we propose a learning approach where users provide examples of transformed data such as those shown in the second column in Figure 1. Our approach does not require users to teach the system step by step and does not require users to know the transformation language. Users simply select a value they want to transform and edit it to represent it in the desired format. The system uses the transformed value to learn a transformation program consistent with the example and applies it to the rest of data. Users then review these transformed results. If there are incorrectly transformed results, the user provides a new example for one of the incorrect entries. The system uses the newly provided example to re-learn a transformation program and apply it to the rest of data. Users repeat this process until satisfied with all the transformed results.

For example, as shown in Figure 2, the user wants to put the city name at the beginning and put a comma between the city name and street address. The user first provides an example "London,1 Lombard Street" for the original string "1 Lombard Street,London". The system then performs three

*This research is based upon work supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Air Force Research Laboratory (AFRL) contract number FA8650-10-C-7058. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, AFRL, or the U.S. Government.

IIWEB '12 Scottsdale, AZ USA

Figure 2: User provides examples for the system to learn transformation programs

Before	After
1 Lombard Street,London	London,1 Lombard Street
1 Dominick Street,New York	New York,1 Dominick Street

Before	After
020-7231 8808	(020)7231 8808
097-3263 8848	(097)3263 8848

Figure 1: Two data transformation scenarios

steps. In the first step, it generates several transformation programs that are consistent with the examples and applies them to the entire dataset. In the second step, the system ranks the transformation programs based on an analysis of the transformed data, preferring rules that leave the data set in a more homogenous state. In the third step, the system shows the transformed data for the top K transformation programs (our figure shows the top 1). The user can select one as the correct and final one or provide a new example. In our example, the user sees that the result for "1 Dominick Street,New York" is not correct (panel 2) and provides a new example "New York,1 Dominick Street" for the system to relearn the transformation program (panel 3). After the second example, all data is transformed correctly (panel 4), and the user accepts the learned transformation.

Learning transformation programs from examples is challenging because in general it is possible to generate many transformation programs consistent with a small number of examples. For instance, the following two transformation programs are consistent with the first row in Figure 1:

- Move the 7th token to position 0 and move the "," to position 2
- Move the token after "," to position 0 and move the "," to position 1

When applying the above transformations to the second row shown in Figure 1, the transformed results would be

- New,1 Dominick Street York
- New, York1 Dominick Street York

These transformed results are obviously wrong. The reason is that some of the interpretations consistent with one example or several examples may not be applicable to other examples. Moreover, as users only provide a few examples, these cannot represent all the variations in the data, so some transformation programs that are consistent with all examples might not able to transform all test data correctly. We face two technical challenges: 1. Identifying as many transformation programs as possible that are consistent with all examples. 2. Ranking all the consistent transformation programs to identify the correct transformation programs.

In order to solve the above challenges, we present an approach to infer correct transformation programs from a few given examples that can transform both training and testing data correctly. In the evaluation section, we test our system on a representative set of data transformation scenarios. The results show that our approach in most cases can generate correct transformation programs for all editing scenarios within 3 examples, can often rank the correct transformation program as the top choice, and the ranking is significantly better than random.

2. PROBLEM

We define the set of transformation programs that the system can generate using a grammar (Figure 3). A transformation program consists of an arbitrary number of 3 kinds of editing operations (line 0), namely insert, move and delete. Line 1 specifies that one insert operation contains 2 pieces of information: the tokens to be inserted and the place where they should be inserted. Line 2 specifies that move operations need to know what elements to move and where to move them to. Line 3 specifies that deletion operations need to know what elements to delete. The term "tokenspec" in line 6 specifies the target subtoken sequence that a user wants to edit. It could be specified by using the type of the token or the actual content of the token. As in Figure 2, if we want to move "London", the "tokenspec" could be "London" or WRDTYP (word type token). The target token sequence can also be specified based on position using "range," which has a start and end position. The actual position is decided by "positionquantifier," which can be a landmark or an absolute position. For the above example, the start position of "London" specified by "posquantifier" could be either 0 or ",", which is a landmark used to indicate the start of city name. The term "where" is used to specify the operation's destination, which has a similar definition to start or end. This transformation grammar is rich, but it is not universal. Our goal in this phase of the work was to validate our approach using a non-trivial grammar.

We formulate our problem as follows. Let $G = (ins|mov|del)*$ be the set of possible transformations. $E = \{(s_i, s_i^{'})|s_i \in S, s_i^{'} \in D\}$ is the definition of the given examples; $(s_i, s_i^{'})$

```
0  program → (ins | mov | del) +
1  ins → INS (token)+ where;
2  mov → MOV tokenspec where ∨ MOV range where
3  del → DEL what ∨ DEL range
4  what → quantifier tokenspec;
5  quantifier → ANYNUM ∨ NUM ;
6  tokenspec → singletokenspec ∨ singletokenspec tokenspec ;
7  singletokenspec → token ∨ type ∨ ANYTOK;
8  type → NUMTYP ∨ WRDTYP ∨ SYBTYP ∨ BNKTYP ;
9  range → start end; scanningOrder → FRM _BEG ∨ FRM _END;
10 start → scanningOrder posquantifier ;
11 end → scanningOrder posquantifier ;
12 where → scanningOrder posquantifier ;
13 posquantifier → INCLD? tokenspec ∨ NUM ;
```

Figure 3: Transformation grammar used to describe common editing operations

represents one example; S is the set of all original values and D is the set of all values. T, the set of transformations that are consistent with examples is defined as $T = \{t \in G | t(s_i) = s_i^{'}\}$. Given a ranking function $m(S, S^{'})$ based on the original data S and the transformed data $S^{'}$, our data transformation problem can be formally defined as

$$t^* = \underset{t}{argmin}\{m(S, S^{'})\ S^{'} = t(S), t \in T\} \quad (1)$$

3. APPROACH OVERVIEW

We decompose our problem into two parts. In the first part we develop efficient algorithms to compute the set T of transformations consistent with the examples provided. In the second part we propose a learning approach to specify the ranking function m.

3.1 Search Space

The search space for transformation programs is $(ins|mov|del)^*$. Without loss of generality we refactor this space into $(ins)^*(mov)^*(del)^*$. The insert phase $(ins)^*$, inserts tokens that appear in the target token sequence but were not part of the original token sequence. The move phase $(mov)^*$ reorders the tokens so that they appear in the same ordering as in the target sequence. The delete phase $(del)^*$ removes tokens that do not appear in the target token sequence. We do insertions first so that the movement and deletion phases can use the inserted tokens as landmarks. We do deletions last so as to not remove landmarks that can be used to specify other operations.

In order to reduce the size of the search space, we introduce subgrammar spaces, which are a disjunction of restricted versions of the whole grammar space. Each subgrammar space is defined by a particular sequence of edit operations. We use the following algorithm to generate subgrammar spaces.

- Step 1: Tokenize the original and target strings and add special start and end tokens.
- Step 2: Generate alignments between examples.
- Step 3: Generate a sequence of edit operations for each example using the alignments.
- Step 4: Identify correspondences among editing sequences of different examples.
- Step 5: Identify the grammar non-terminals that cover the common values across examples.

In step 1, our approach tokenizes all the examples. For example "1 Lombard Street,London" is tokenized as START() NUMTYP(1) BNKTYP( ) WRDTYP(Lombard) BNKTYP( ) WRDTYP(Street) SYBTYP(,) WRDTYP(London) END(). NUMTYP, BNKTYP, WRDTYP and SYBTYP represent different token types (number, blank, word or symbol). The START() and END() identify the start and end of the original token sequence.

In step 2, our approach uses a simple alignment algorithm to identify the same tokens in the original and target token sequences (e.g., the token "London" in Figure 1). When tokens appear multiple times, the algorithm generates all possible alignments.

In step 3, we generate a set of possible edit operation sequences from each alignment. For the pair "1 Lombard Street,London" and "London,1 Lombard Street", the alignment algorithm determines that all tokens in the target token sequence can be mapped to the original token sequence. Consequently, only move operations are needed. In the move phase, the alignment results would indicate the new position of the token NUM(1) is behind WRDTYP(London). A mov operation such as mov(7,9,0) would be generated, which means moving the subtoken sequence located within position 7 and 9 to position 0. In the delete phase, the algorithm deletes the START and END tokens.

In step 4, in order to generate data transformations that cover all examples, we cluster the edit operations derived from the same transformation program. We cluster together the edit sequences with the same length and edit operator type of different examples. For our scenario shown in figure 1, one possible cluster is mov(7,7,0), mov(7,7,1), del(2,2), del(8,8) for the first example and mov(7,9,0), mov(6,6,3), del(4,4), del(10,10) for the second example.

In step 5, we generalize the operator sequences from multiple examples to identify the candidate values for a set of non-terminals of each example. Then we intersect these candidate values to produce the candidate values for this subgrammar space. Based on the result of step 4, the non-terminal "tokenspec" (Figure 3, line 6) would have an empty value set as WRDTYP(New) BNKTYP( ) WRDTYP(York) and WRDTYP(London) cannot be generalized in the grammar.

Finally, the original grammar space $(ins|mov|del)*$ has been reduced to $(mov_1^1 mov_2^1 del_3^1) \vee (mov_1^2 mov_2^2 del_3^2 del_4^2) \vee \ldots$.

3.2 Search for Consistent Programs

To identify consistent transformation programs, we search for consistent programs in the subgrammar spaces. Figure 4 illustrates the idea. Different rectangles represent different subgrammar spaces, where search trees are built in these subgrammar spaces. Each internal node of the search tree represents the current token sequence and the edge represents the transformation rule generated from an editing component such as mov_i^j. The goal is to identify the path that leads from the original token sequence to the target token sequence. However, this search space is huge. The example shown in Figure 2, whose minimal edit sequence is of length, generates 50 subgrammar spaces, and the steps generate over 400 rules. The total number of possible transformation programs would be $400^3 * 50$. Actually in many other scenarios, the length of the edit sequence exceeds 10

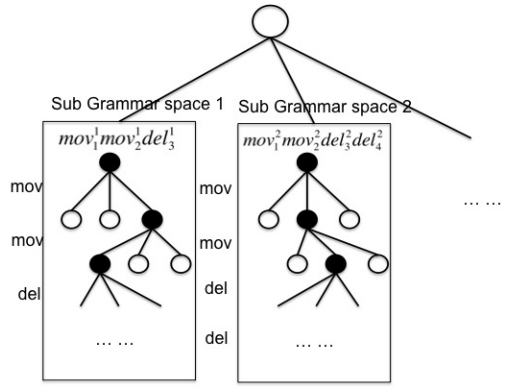

Figure 4: Using the UCT algorithm to identify consistent transformation programs

and the number of subgrammar spaces exceeds 1000. It is impossible to consider all possible transformations.

We use a sampling-based search algorithm to identify transformation programs (editing path) in these spaces. The algorithm consists of following steps:

- Choose a subgrammar space to explore.
- Repeatedly use the *searchOneTime* algorithm within the chosen subgrammar to identify consistent programs.
- Repeat this procedure a predefined number of times.

searchOneTime Algorithm.

Our searchOneTime algorithm uses the UCT algorithm[5], an extension of the UCB algorithm[1], to balance the exploration and exploitation in a large search space. It has been successfully applied to the Computer GO game [3]. The basic idea is as follows. As each step has a huge number of branches to explore, it is impossible to go through each branch to identify the right path in the search tree. Only a small number of branches are sampled during each step to be evaluated and be explored further. The basic idea of UCT is to explore the branches that have a known high score frequently, but also visit the rarely visited nodes. We use the UCT algorithm to perform a one time search in one subgrammar space (Figure 5).

```
function searchOneTime(state)
    if terminal(state):
        return getUCBScore(state)
    endif
    ruleSet = sampleRules(state)
    for each rule_i of ruleSet:
        newState_i = applyRule(rule_i, state)
        newState_i.value = getUCBScore(newState_i)
    endfor
    mState = the state with minal value of all children state
    mState.value = searchOneTime(mState)
    update(state, minState)
    return state.value
end function
```

Figure 5: UCT Algorithm

sampleRules() is used to randomly sample a set of rules based on the current state. *applyRule()* applies $rule_i$ on the current state leading to a new state. *getUCBScore*, the scoring function defined in Equation 2 is used to evaluate these new states. *update()* uses the value returned by *mState* as the root's value. Finally, the algorithm selects the child state with minimal score to explore.

$$I_t = \underset{i}{argmin}\{\overline{V}_{i,T_i(t-1))} - \theta * c_{t-1,T_i(t-1)}\}, c_{n,m} = \sqrt{\frac{ln(n)}{5m}} \quad (2)$$

This function is used to evaluate children states to identify one that has the minimum value. t is the number of times that the parent state has been explored. $T_i(t-1)$ indicates the number of times that the i^{th} child has been explored. $\overline{V}$ is the average heuristic value of the i^{th} child, averaged over the $T_i(t-1)$ times that it has been explored. θ is a coefficient used to adjust the weight of c, the term used to balance the exploitation and exploration. c grows when the parent has been visited often and the child has only been visited few times. It also means that this child becomes more likely to be explored next time.

Heuristic Values.

Heuristic values, which correspond to the $V_{i,T(t-1)}$ shown in Equation 2 are used to evaluate the subgrammar space nodes and internal search tree nodes. As these two kinds of nodes are different, we use different evaluation functions for them. The heuristic used to evaluate the subgrammar space is listed in Equation 3.

$$V_{i,t} = \frac{\overline{s}_{i,T(t)}}{l_i * Z}, Z = \sum_{i}^{k} V_{i,t} \quad (3)$$

$\overline{s}_{i,T(t)}$ is the average value for the initial state. l_i is the length of edit sequence that is generated from the i^{th} subgrammar space where the shorter edit sequence is preferred as it provides a more concise way of editing. Z is a normalizing term. The heuristic used to evaluate internal search tree nodes is a normalized version of edit distance between the current token sequence and the target token sequence.

3.3 Transformation Ranking

The previous step can generate many transformation programs given that many transformation programs are consistent with a small number of examples. The idea for ranking them is based on analyzing their effects on the entire data set. Consider the second scenario in Figure 1, if we just use the first row as an example, two transformation programs are consistent: The first one: "insert the parentheses at 0 and 2, delete the token after parenthesis and before the number," produces Result 1, below. The second one: "insert the parentheses at 0 and 2, delete the token after the parenthesis and before 7231," produces Result 2, which fails on the second row and third row.

Result 1	Result 2
(020)7231 8808	(020)7231 8808
(097)3263 8848	(097)-3263 8848
(020)7297 5600	(020)-72975600

Even though Result 2 may well be what the user wants, we would like to rank Result 1 higher because the result is more homogeneous. We implement this heuristic using a logistic regression classifier to classify the transformed results as regular or irregular. We measure the homogeneity of the transformed data using a set of features that depend on all the rows of the data set. These features belong to three categories. The first category captures the homogeneity of inserted tokens by calculating the entropy of the count of

inserted tokens. The second category captures the homogeneity of the reordered tokens by calculating the entropy of the count of different move directions. The third category captures the homogeneity of the deleted tokens by calculating entropy of the count of deleted tokens. In our example we have a feature that computes the entropy of the deleted hyphen token. In Result 1, the number of hyphen tokens deleted in each row is 1, 1, and , so the entropy is 0. In Result 2, the hyphen was not deleted in the second and third rows, so the number of deleted hyphen tokens is 1, 0, and 0, so the entropy is larger than 0.

We rank the transformation programs using the confidence of the logistic regression classifier that classifies the results as regular or irregular. Unless there is a clear winner, we show users the top K results. Our hypothesis, to be tested in future user studies, is that users will be able to quickly see which result is correct or closer to what they are expecting. If the results are incorrect, the expectation is that examples provided on the results that are closer to what users want will enable the system to more quickly converge on the desired transformation program.

4. RELATED WORK

Data transformation used in metadata and data warehouse means converting data from a source data format into desired data. The most closely related work is Data Wrangler [4]. It uses the transformation language defined in Potter's Wheel [7] and provides a nicer GUI, which provides several transformation suggestions based on user input. Users can specify desired transformations by editing examples, selecting from suggested rules, or transforming rules using menus. However, both Data Wrangler [4] and Potter's Wheel [7] require users to provide examples step by step, which usually requires a user to give a sequence of examples for a simple transformation scenario. Liang et al. [6] aims to learn the text editing program through examples. They adopt a multitask learning model to take advantage of shared subtasks to identify the most probable edit programs. The training data used in their approach are also edit sequences, which consists of the intermediate steps for a transformation task.

Tuchinda et al. [8] present a Mashup by demonstration system. It integrates extraction, cleaning, integration and display together into one system that allows a user focus on the data rather than the process. The data cleaning step referred in this paper uses examples to select transformations from a library of predefined rules. As there are many types of possible transformations, it is hard to build a comprehensive library of predefined rules. Our approach builds the rules from the examples. Google Refine[1] supports both data cleaning and data transformation. Users can easily specify common transformations using menus (e.g., split fields based on separators), but they must specify more complex transformations using its scripting language.

Christen et cl. [2] present a name and address cleaning and standardization approach to convert the data into its standard form using a correction list. These data are then tagged using a lookup table and then segmented into different fields using semantic labels in a HMM model. It is different from our work as it only aims to solve the address and name normalization through training a HMM model on large training datasets.

[1]http://code.google.com/p/google-refine/

Table 2: Average number of examples required to generate correct transformation programs

Dataset	Example Count	Correct TPs
address1	1.25	33.5
address2	5.25	3.75
date1	1	2
date2	1.5	3.5
tel1	1	223
tel2	1	60.75
time	2.5	1.75

5. EXPERIMENTS

In this section, we introduce two experiments that provide strong evidence that our approach can identify consistent transformation programs and rank them effectively.

Our experimental data set consists of 6 transformation scenarios. Each scenario has 50 rows of data containing both the original string and its corresponding transformed value. These scenarios with their descriptions are shown in Table 1.

5.1 Identifying Correct Programs

In this experiment we simulate the user's behavior. We first choose one example randomly and learn a set of transformation programs consistent with it. Then we evaluate these transformation programs on all the data. If all these transformation programs produce at least one incorrect result, we add to the set of examples an entry corresponding to a randomly chosen incorrect result. We repeat this process iteratively until the system finds a transformation program that transforms both the training data and test data correctly. The parameters for the search algorithm are set manually. The search depth is set to 6 and the number of rules sampled for each state is set to 10. 50% of the total number of subgrammar spaces are sampled and explored. For each sampled subgrammar space, we run the search algorithm 200 times.

Table 2 shows the results of running this experiment 20 times. The table shows the average number of examples required to learn a correct transformation program (Example count) and the average number of correct transformation programs found (Correct TPs).

The number of examples required to infer a correct transformation program is small in all scenarios except for address2, which required on average 5.25 examples. In this scenario it is important to choose examples with different number of words in the city name. Random selection of examples may choose examples that eliminate programs that are too specific (e.g., move the token "London"). Choosing an example with a different number of words would rule out programs that do not specify the *what* part based on an anchor as required to transform the data correctly. Such an example would also rule out the over-specific programs that contain a single word in the city name. We plan to investigate active learning algorithms capable of suggesting the items that users should edit to provide new examples.

5.2 Ranking Results

To test our ranking algorithm, we first use our search algorithm to compute a set of transformation programs con-

Table 1: Dataset Description

Dataset	First Row	Description
address1	(Brankova 13 , Brankova 13)	Replace with a blankspace
address2	(1 Lombard Street,London , London,1 Lombard Street)	Move city name to front; put comma in the middle
date1	(2010-07-30 , 07/30/2010)	Reverse the order and replace the hyphen with slash
date2	(13/05/2010 , 2010-05-13)	Put month & date in front, replace hyphen with slash
tel1	(Tel:</B> 020-7928 3131 , 020-7928 3131)	Remove the prefix
tel2	(020-8944 9496 , (020)8944 9496)	Add parenthesis around first 3 digits; remove hyphen
time	(1 January 2007 4:48pm , January 1,2007 4:48pm)	Change the order of first 3 tokens and insert a comma

sistent with the minimum number of examples required to compute at least one correct transformation program. Then we use our ranking algorithm to rank all the transformation programs consistent with these examples.

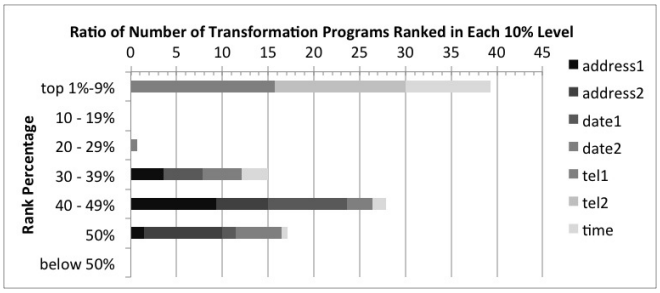

Figure 6: Ranking results.

Figure 6 shows the results of running this experiment 20 times. For each scenario we count the number of times that our ranking algorithm places the correct transformation program in the top 9%, in the 10 to 19%, etc. Each bar in the figure shows this count normalized by the number of consistent transformation programs in each run. Our ranking algorithm is able to place the correct transformation program in the top 10% of consistent transformation programs close to 40% of the time. Moreover, the ranking algorithm always places the correct program in the top 50%, outperforming random guessing. The experiment also shows that the characteristics of the data set significantly affect the effectiveness of the ranking algorithm. The algorithm performs poorly in the address2 scenario, which consists only of move operations where our features cannot detect that a multi-word city name is not moved as a unit. The algorithm performs well in scenarios that feature a mix of operations.

6. CONCLUSION

This paper introduced a data transformation approach. It learns data transformations from user-provided examples. We define a grammar to describe common user editing behaviors. Our approach then reduces the larger grammar space to a disjunction of subgrammar spaces using examples. We apply a UCT-based search algorithm to identify consistent transformation programs in these subgrammar spaces. To handle the many interpretations, we use a transformation ranking method to identify the correct transformation programs. Our experimental results show that in most cases our approach generates a correct transformation program using fewer than three examples, ranks all correct transformation programs within the top 50%, and 40% of the time ranks the correct transformation program in the top 10% of transformation programs consistent with the examples.

7. REFERENCES

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[2] P. Christen, T. Churches, and J. X. Zhu. Probabilistic name and address cleaning and standardisation. In *Proceedings of the Australasian Data Mining Workshop*, 2002.

[3] S. Gelly and D. Silver. Achieving master level play in 9 x 9 computer GO. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 1537–1540. AAAI Press, 2008.

[4] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 3363–3372, 2011.

[5] L. Kocsis and C. SzepesvÃąri. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[6] P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*, 2010.

[7] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 381–390, 2001.

[8] R. Tuchinda, C. A. Knoblock, and P. Szekely. Building mashups by demonstration. *ACM Transactions on the Web (TWEB)*, 5(3), July 2011.