Programming by Demonstration with DLProlog

José Oscar Olmedo-Aguirre[1], Marisol Vázquez-Tzompantzi[2], Giner Alor-Hernández[3]

[1] Cinvestav-IPN, Department of Electrical Engineering, Mexico City, Mexico

[2] Cinvestav-IPN, DCTS, Mexico City, Mexico

[3] Instituto Tecnolgico de Orizaba, Orizaba, Veracruz, Mexico

oolmedo@cinvestav.mx, galor@itorizaba.edu.mx, mvazquezt@cinvestav.mx

Abstract. Traditionally computer programming has been conducted as a long lasting cycle of coding, testing and debugging. In comparison, in programming by demonstration (PBD), a system automatically attempts to construct a program that produces the same input-output pairs given as examples by a demonstrator. Unfortunately, in most PBD systems this automatic program construction process has been approached by finding the proper composition of simpler functions searched from a basic set. The exponential computational costs of such searches makes this approach impractical except for a few domains (i.e. text processing) in which some demonstrator's clues can be used to dramatically reduce the search space. In this work, some of the features of a PBD system under construction are presented that does not rely on function composition but in the step by step construction of a program automaton. The automaton is described in DLProlog, a dynamic logic modal extension of pure Prolog. DLProlog allows to represent the automaton states as user defined first-order predicates and the automaton transitions as logic rules (i.e. dynamic logic modal Horn clauses). The main contributions of this work are twofold: (i) by using a spreadsheet like system for conducting the demonstrations where the basic behavioral rules of the program are learnt by the system, and (ii) by using a dymanic logic modal inference system to construct complex behavioral program rules from simpler ones. Besides the system not only attempts to construct the program but also assists on providing the formal specifications of the input-output conditions, an aspect of paramount importance for program development.

Keywords: Programming by example, logic programming, dynamic logic, knowledge representation and reasoning, machine learning.

1 Introduction

According to [2], in programming by demonstration (PBD), the demonstrator should be able to instruct a system to learn that demonstrator does and the system should create the program that reproduces the users actions. In comparison, in programming by example (PBE) [2, 3], an end-user demonstrates to a system a set of examples consisting of input-output pairs and from them the system will attempt to infer the program that produces these input-output relations. While in PBD the demonstrator provides more guidance to the system to learn the intended program through the observed actions, in PBE the demonstrator only provides a concrete list of input-output pairs from which the system will accomplish the challenging task of deriving the entire program. Hence PBD is intended for programmers with varying degrees of experience, whereas PBE is destined for end-users with no interest in programming the task at hand.

Both programming paradigms are very close areas of research in machine learning, though the more ambitious goals of PBE comes with a price. As argued by Adytia el al. [4], there are two major concerns in this approach: firstly, the very few examples given per task by the demonstrators, and secondly, the searching over arbitrary composition of consistent candidate functions. Possibly, the second one is more challenging because the function composition problem is a NP-complete problem and defies all the state-of-the-art search techniques due to the fact that any change in the input-output relation may radically change the entire program. As an example of the unstable behavior of a PBE system, consider the simple task of reversing lists. As training examples for the PBE system, the demonstrator provides the list of input-output pairs shown in Figure 1:

Input	Output
[ ]	[ ]
[a]	[a]
[6]	[6]
[a, b, c, d]	[d, b, c, a]
[0, 1, 2, 3, 4, 5, 6]	[6, 5, 4, 3, 2, 1, 0]

Fig. 1. A list of input-output pairs in the reverse program demonstration.

With no additional information, the PBE system may infer that the output is produced from the input by applying the reverse sorting function. Nonetheless, by including the input [a, b, e, c, d] as testing case, a wrong output is produced as shown next:

Input	Wrong Output	Expected Output
⋮	⋮	⋮
[a, b, e, c, d]	[e, d, c, b, a]	[d, c, e, b, a]

where the reverse sorting function that produced the wrong output needs to be discarded because it does not lead to the expected output. In some problem domains like text processing, the user may provide in the examples additional information that the PBE system uses to draw as many clues as possible. The clues are used in heuristics that may help to reduce the vast search space by selecting better interpretations of the examples. Fortunately, because PBE systems are mainly used in the user-interface, for example of a spreadsheet program, their close interaction with end-users allows the system to acknowledge when the function composed is incorrect whenever the output produced by the system is erased by the user. With this action the PBE system starts searching again for another function that can reproduce the new example along with all the others.

However, the price of this form of unsupervised machine learning of a PBE system is high and not required when the demonstrators are not end-users but programmers. Programmers do not need a system that guess functions that may be wrong for the problem at hand. That programmers need is a programming environment that simplifies the demonstration and where the programmers be able to undertake some abstract reasoning in order to generalize to a parameterized program from the concrete examples provided.

The PBD under design rest on the following principles observed on the programmers behavior when dealing with complex programs:

- Programmers prefer a short cycle of interaction with the input-output pairs obtained by the program. They prefer to find errors as soon as possible in their understanding of the problem rather than spending too much time in a program that may not produce the expected results. Generally, the time spent by programmers to know if their code meet their expectations (not to say specifications) long last in a compiled program than in an interpreted one.
- Programmers find easier to deal with concrete examples instead of going directly through the program codification. Very often programmers need to make some small tests and experiments in order to make sure they understand how to build the program correctly. Unfortunately, the problem with this approach is in the selection of the input data, because they may not consider all the cases in the program design that are required to take into account. However, if the selection exhaustively considers all the relevant cases, this strategy is very helpful to gain deep insights not only on the program design but also on solving the problem at hand.

These principles have motivated the development of a PBD environment that uses DLProlog [8] as its foundational programming language. DLProlog provides the programming concepts used in the environment like program state and program state transition represented by predicates and logic rules (i.e. dynamic logic modal clauses), respectively.

This article is organized as follows. In section 2, a succinct review of the related work is presented. In section 3, a demonstration of how to reverse a finite

list of elements is given. In section 4, the clause learning process is presented in some detail. In section 5, the program construction process is presented as the outcome of an inference procedure. Finally in section 6, some concluding remarks are given along with the future work that already is in progress.

2 Related Work

The history of PBD can be traced back as early as the middle of 1970s with the pioneering work of D. Canfield Smith with his Pygmalion [2] system written in SmallTalk. Pygmalion was intended to change the way of programming in which programs are demonstrated concretely to the computer. One of the main concerns introduced by Canfield is about the human-computer interaction dimensions of a programming language and a programming environment. He argues that writing static language statements interleaved with the computer enactment of such statements becomes a poor way of communication when the translation last long periods of compilation and debugging in opposite directions. Upon this approach of programming, Pygmalion was designed as a graphical programming environment that allows users to sketch their program ideas instead of typing program instructions, in a way that the sketches can actually be executed by a computer. After this pioneering work, a number of research proposals came into the scene [2], like Thinker for programming learning, Rehearsal World for developing educational software, Peridot for creating user interfaces, Chimera for graphical editing, The Geometry Sketch Pad for problem solving in Euclidian Geometry, and TELS for text editing tasks, among others. Though all of them have a notorious influence in modern graphical user interfaces, TELS have silently influenced the text entry systems of our modern intelligent mobile phones.

However, the interest for PBE and PBD spans over fields as different as diverse like Robotics where the purpose is to teach to robots how to achieve some repetitive tasks by user demonstrations [5]. Despite of nowadays there is no a successful and widely used PBE environments, some techniques have found their place and are of common use in widely accepted tools like spreadsheets and graphical user interfaces that learns from user inputs of text and predicts the next input [4]. Because of their theoretical and practical importance, PBE and PBD are important research areas in the AI field of Machine Learning [6].

3 Programming by Demonstration

The PBD system under construction provides a programming environment with a spreadsheet as a front-end user interface along with a small set of basic list processing functions like hd(), tl() and cons(). Assuming that xs denotes a non-empty list, function hd(xs) returns the first element of xs and tl(xs) returns the remaining list after removing the first element of xs, whereas cons(x, xs) written $[x|xs]$ as in Prolog constructs a list with x as its first element and xs as a list with the remaining elements. The purpose of the environment is to provide an

electronic pad where the programmer can figure out the sequence of operations the program should follow to achieve the task. Figure 2 shows a demonstration of how to reverse a list in the PBD environment.

State	xs	ys_1	ys_2	zs
Reverse_0	[a, b, c, d]	–	–	–
Reverse_2	[a, b, c, d]	[ ]	[a, b, c, d]	–
Reverse_2	[a, b, c, d]	[a]	[b, c, d]	–
Reverse_2	[a, b, c, d]	[b, a]	[c, d]	–
Reverse_2	[a, b, c, d]	[c, b, a]	[d]	–
Reverse_2	[a, b, c, d]	[d, c, b, a]	[ ]	–
Reverse_1	[a, b, c, d]	–	–	[d, c, b, a]

Fig. 2. Demonstration of how to reverse a list.

The demonstration that reverses a list is very simple though illustrative. It proceeds by splitting the input list xs into two lists ys_1 and ys_2, where ys_1 contains the elements of xs already reversed and ys_2 contains the remaining elements of xs not reversed yet. Thus, for example, the bindings $xs = [\text{a}, \text{b}, \text{c}, \text{d}]$, $ys_1 = [\text{b}, \text{a}]$ and $ys_2 = [\text{c}, \text{d}]$ describe a valid state during the course of the reverse action on xs. For the next valid state, the first element of ys_2 is removed and inserted at the front of ys_1. In this way, the list ys_2 of remaining elements is decreased one by one and the list ys_1 of reversed elements is increased. When eventually, the list ys_2 becomes empty, the list ys_1 becomes the reverse of xs. As the functions app() and rev() need to be defined for the specifications and for the proof of correctness, the several technical aspects in relation to the structural properties of lists cannot be explained here in detail for lack of space. From the usual pure Prolog definitions of reverse/2 and append/3, the definitions of functions app() and rev() can be established as follows:

$$\text{app}(xs, ys) = zs \Leftarrow \text{append}(xs, ys, zs)$$
$$\text{rev}(xs) = zs \Leftarrow \text{reverse}(xs, zs)$$

From the demonstration, the system will try to identify the minimal set of states that the reversing program requires. By representing states as predicates, the signature of each entry of the table helps to identify each state. The signature is the list of types of the variables that the state holds. Thus the signature of states $\text{Reverse}_0(xs)$ and $\text{Reverse}_2(xs, ys_1, ys_2, zs)$ are respectively List and (List,List,List,List). In Figure 3, the states identified from the demonstration are presented along with their invariant conditions as their definitions.

The invariants are the valid conditions that always hold for the values of the variables bound to them at the corresponding states. Intuitively the invariants ensure that the program only transits among valid states. Therefore, the invariants can be used to state the partial correctness of the program where the initial state $\text{Reverse}_0(xs)$ stands for the precondition and the final state

$$\begin{array}{ll} \text{Reverse}_0(xs) & \Leftarrow \text{List}(xs) \\ \text{Reverse}_1(xs, zs) & \Leftarrow \text{rev}(zs) = xs \\ \text{Reverse}_2(xs, ys_1, ys_2) & \Leftarrow \text{app}(\text{rev}(ys_1), ys_2) = xs \end{array}$$

Fig. 3. States identified from the demonstration.

$\text{Reverse}_1(xs, zs)$ stands for the postcondition. Currently, the demonstrator must provide definitions for the invariants if the user wants to formally proof partial correctness. However, for the derivation of the program, the invariants are not required.

4 DLProlog Clause Learning

The following tables describe the simple rules of computation that the system have learnt from the demonstration. The structure of these tables is similar to the structure of the table used in the demonstration, though restricted to only three rows. The first row shows the name of the rule and the variables with one column for each variable. The second row shows the name of the state at the precondition, the values that each of the variables take at this state, and the guard condition for the rule. The third row shows the name of the state at the postcondition, the values that each of the variables take at this state obtained from the precondition by application of any available functions. For the precondition, the last column shows the guarding condition that selects the valid values to which the rule can be applied. For the postcondition, the last column shows the action, generally an assignment of values taken from the precondition to the variables shown in the postcondition.

In Figure 4, rule R_1 takes the state $\text{Reverse}_0(xs)$ with $xs = [\text{a}, \text{b}, \text{c}, \text{d}]$ as precondition and the state $\text{Reverse}_2(xs, ys_1, ys_2)$ with the values [ ] and [a.b.c.d] bound to the variables ys_1 and ys_2, respectively, as postcondition. Because this rule always applies at the beginning of the computation, the program transits unconditionally from state Reverse_0 to state Reverse_2. In order to infer this rule, at state Reverse_0 the system generalizes the values that can take the input parameter xs to lists with arbitrary but finite number of elements. Then, because at state Reverse_2 the demonstrator puts the empty list into variable ys_1 and copies the values of xs into variable ys_2, the system infers that the action consists of the multiple assignment that perform this task as shown in the rule. Below the table, the rule R_1 is shown written in DLProlog.

In Figure 5, rule R_2 takes the state $\text{Revese}_2(xs, ys_1, ys_2)$ with $xs = [\text{a}, \text{b}, \text{c}, \text{d}]$, $ys_1 = [\text{d}, \text{c}, \text{b}, \text{a}]$ and $ys_2 = [\,]$ as precondition and the state $\text{Reverse}_1(xs, zs)$ with xs as before and $zs = [\text{d}, \text{c}, \text{b}, \text{a}]$ as postcondition. This rule only applies in the case that $ys_2 = [\,]$, when there are no more elements in the list to reverse. In this case, the value assigned to the variable ys_1 is simply copied to the output variable zs. Because Reverse_2 is the final state of the automaton for the reverse program, there are no rules having Reverse_2 as precondition. In consequence there are no more rules that can be applied and then the program stops. Having

	xs	ys_1	ys_2	zs		
Reverse_0	[a, b, c, d]	–	–	–	true	(a)
Reverse_2	[a, b, c, d]	[ ]	[a, b, c, d]	–	(ys_1,ys_2):=([ ],xs)	

$$[(ys_1,ys_2):=([\,],xs)]\,\text{Reverse}_2(xs, ys_1, ys_2) \Leftarrow \text{Reverse}_0(xs) \quad \text{(b)}$$

Fig. 4. Rule R_1 (a) condition-action table, (b) DLProlog clause

the program stopped at this state, the postcondition of the reverse program becomes necessarily satisfied because the postcondition is logical consequence of the state Reverse_2.

	xs	ys_1	ys_2	zs		
Reverse_2	[a, b, c, d]	[d, c, b, a]	[ ]	–	$ys_2 = [\,]$	(a)
Reverse_1	[a, b, c, d]	[d, c, b, a]	[ ]	[d, c, b, a]	zs:=ys_1	

$$[zs{:=}ys_1]\,\text{Reverse}_1(xs, zs) \Leftarrow ys_2 = [\,], \text{Reverse}_2(xs, ys_1, ys_2) \quad \text{(b)}$$

Fig. 5. Rule R_2 (a) condition-action table, (b) DLProlog clause.

In Figure 6, rule R_3 takes the state $\text{Reverse}_2(xs, ys_1, ys_2)$ with xs = [a, b, c, d], ys_1 = [a] and ys_2 = [b, c, d] as precondition and the same state as postcondition although with ys_1 = [b, a] and ys_2 = [c, d]. This rule only applies in the case that $ys_2 \neq [\,]$, when there are still some (at least one) elements to be reversed. In the demonstration, there are a number of actions to be taught to the system by: (i) passing the mouse over the list ys_2 that makes selectable both the first and the rest of this list, (ii) selecting the first element of ys_2, (iii) dragging the selected element to the front of ys_1 and dropping it there, (iii) passing the mouse over the list ys_2, (iv) selecting the rest of the list ys_2, and (iii) dragging the selected list to ys_1 and dropping it there. These actions can be summarized in the single DLProlog action $(ys_1, ys_2){:=}([\text{hd}(ys2)|ys_1], \text{tl}(ys_2))$ that appears at the modality in the head of the clause R_3.

	xs	ys_1	ys_2	zs		
Reverse_2	[a, b, c, d]	[a]	[b, c, d]	–	$ys_2 \neq [\,]$	(a)
Reverse_2	[a, b, c, d]	[b, a]	[c, d]	–	(ys_1,ys_2):=([hd(ys_2)|ys_1],tl(ys_2))	

$$[(ys_1,ys_2):=([\text{hd}(ys_2)|ys_1],\text{tl}(ys_2))]\,\text{Reverse}_2(xs, ys_1, ys_2) \Leftarrow ys_2 \neq [\,], \text{Reverse}_2(xs, ys_1, ys_2) \quad \text{(b)}$$

Fig. 6. Rule R_3 (a) condition-action table, (b) DL Prolog clause.

DL Prolog rules R_1, R_2 and R_3, along with R_0 (not presented here by its simplicity) that were learnt by the system from the reverse program demonstration are ready to be used for the synthesis of a rule with a single compound action that has the behavior expected from the reverse program.

5 Automated Program Construction

The reverse list rules learnt from the demonstration, including also rule R_0, are shown next:

$$
\begin{array}{l}
\mathrm{R}_0 : \mathrm{Reverse}_0(xs) \Leftarrow \mathrm{List}(xs) \\
\mathrm{R}_1 : [(ys_1,ys_2){:=}([\,],xs)]\,\mathrm{Reverse}_2(xs, ys_1, ys_2) \Leftarrow \mathrm{Reverse}_0(xs) \\
\mathrm{R}_2 : [zs{:=}ys_1]\,\mathrm{Reverse}_1(xs, zs) \Leftarrow ys_2 = [\,], \mathrm{Reverse}_2(xs, ys_1, ys_2) \\
\mathrm{R}_3 : [(ys_1,ys_2){:=}([\mathrm{hd}(ys_2)|ys_1], \mathrm{tl}(ys_2))]\mathrm{Reverse}_2(xs, ys_1, ys_2) \\
\qquad\qquad \Leftarrow ys_2 \neq [\,], \mathrm{Reverse}_2(xs, ys_1, ys_2)
\end{array}
$$

From these rules, the following rule with a complex action can be constructed:

$$
R : \left[\begin{array}{l}
\texttt{new}\ ys_1, ys_2\texttt{:} \\
\quad \left(\begin{array}{l}
ys_1, ys_2{:=}[\,], xs; \\
\texttt{while}\ ys_2 \neq [\,]\ \texttt{do} \\
\quad ys_1, ys_2{:=}[\mathrm{hd}(ys_2)|ys_1], \mathrm{tl}(ys_2) \\
\texttt{od}; \\
zs{:=}ys_1
\end{array}\right)
\end{array}\right] \mathrm{Reverse}_1(xs, zs) \Leftarrow \mathrm{List}(xs)
$$

Rule R comprises a single compound action that denotes the intended list reverse program. It can be read as follows: given a list xs, after executing the actions enclosed in the brackets, the output list zs is the reverse of the input list xs whenever xs be a list. The above bidimensional arrangement of the program text (by enlarging the enclosing brackets) is preferred here in order to make it more readable in comparison to its actual linear form:

$$
R : [\texttt{new}\ ys_1, ys_2\texttt{:}\ (ys_1, ys_2{:=}[\,], xs; \cdots; zs{:=}ys_1)]\,\mathrm{Reverse}_1(xs, zs) \Leftarrow \mathrm{List}(xs)
$$

where the ellipsis indicates the missing program fragment that can be recovered from the previous program presentation by matching the context in which the ellipsis occur. The program synthesis of the reverse program is the outcome of the inference procedure applied to the rules R_0, R_1, R_2 and R_3. The DL Prolog semantic rules for the automated program construction are presented in Figure 7 in the Gentzen's sequent calculus [7].

Note that the DLProlog rules are written in the forward style $F \Rightarrow [A]F'$. The intuitive meaning of the DLProlog rules of Figure 7 can be outlined as the construction of new program rules from others previously constructed, starting with the learnt rules from the demonstration. Each of these rules introduces a construct as described next.

$$\frac{\mathbf{P}, F \Rightarrow \neg F' \vdash \mathbf{P}'}{\mathbf{P}, F \Rightarrow [\texttt{false}?]F' \vdash \mathbf{P}'} \quad (FI) \qquad \frac{\mathbf{P}, F \Rightarrow F' \vdash \mathbf{P}'}{\mathbf{P}, F \Rightarrow [\texttt{true}?]F' \vdash \mathbf{P}'} \quad (TI)$$

$$\frac{\mathbf{P}, F \Rightarrow F' \vdash P' \quad \mathbf{P} \vdash \mathbf{P}', G}{\mathbf{P}, F \Rightarrow [G?]F' \vdash \mathbf{P}'} \quad (GI) \qquad \frac{\mathbf{P}, F \Rightarrow F'\{x \mapsto t\} \vdash \mathbf{P}'}{\mathbf{P}, F \Rightarrow [x{:=}t]F' \vdash \mathbf{P}'} \quad (AI)$$

$$\frac{\mathbf{P}, F \Rightarrow [A]F', F' \Rightarrow [A']F'' \vdash \mathbf{P}'}{\mathbf{P}, F \Rightarrow [A;A']F'' \vdash \mathbf{P}'} \quad (SI) \qquad \frac{\mathbf{P}, F \Rightarrow [A]F', F \Rightarrow [A']F' \vdash \mathbf{P}'}{\mathbf{P}, F \Rightarrow [A \cup A']\, F' \vdash \mathbf{P}'} \quad (UI)$$

$$\frac{\mathbf{P}, F \Rightarrow F', F' \Rightarrow [A]F', F' \Rightarrow F'' \vdash \mathbf{P}'}{\mathbf{P}, F \Rightarrow [A*]F'' \vdash \mathbf{P}'} \quad (II) \qquad \frac{\mathbf{P}, F \vdash \mathbf{P}', F'}{\mathbf{P}, [A]F \vdash \mathbf{P}', [A]F'} \quad (BI)$$

Fig. 7. DLProlog semantic rules for program construction.

Rule (FI) introduces the modal action false? of testing for falsity by removing the negation from the postcondition $\neg F$. It causes to the current course of actions to fail. Rule TI introduces the modal action true? of testing for validity that casuses no effect in the course of actions. Rule GI introduces the modal action $G?$ of testing for the guard condition G, whenever G can be deducted. Rule AI introduces modal action $x := t$ of the assignment of term t to variable x, if there is a binding of x to t in the substitution $\{x \mapsto t\}$ applied to postcondition F'. Rule SI introduces the modal action $A; A'$ of sequential composition of simpler actions A and A', if the postcondition of A implies the precondition of A'. If any of the actions A or A' fails, the sequential composition fails.

Rule UI introduces the modal action $A \cup A'$ of non-deterministic choice of simpler actions A and A', if they have both the same precondition F and the same postcondition F'. Its intuitive meaning is that it is possible to take either course of actions given by A or A' because both start at F and both terminate at F'. Besides, if none of the actions fail, one of them is chosen non-deterministically; if any of the actions fails, the other takes place; if both fail, then the non-deterministic choice also fails. Rule II introduces the modal action $A*$ of iteration of simpler action A, if its precondition F implies the invariant condition F' of the iteration and the invariant F' implies the postcondition F''. If A fails, the iteration terminates.

Finally rule BI introduces the necessity modal action $[A]$ by modal generalization on formula F, meaning that if F is valid in a state it is also valid by any course of actions A that lead to that state. If the action fails, the postcondition becomes necessarily false.

These semantic rules for the program derivation cannot be applied directly because they are expressed in terms of the basic dynamic logic program combinators $\texttt{false}, \texttt{true}, ;, \cup, *, :=$. The usual structured constucts of DLProlog like `if then else fi` for selection and `while do od` for iteration must be previously translated into the basic combinators by using the following translation schema:

$$
\begin{array}{ll}
\texttt{skip} & \rightarrow \texttt{true}\,? \\
\texttt{fail} & \rightarrow \texttt{false}\,? \\
\texttt{if}\, F\, \texttt{then}\, A\, \texttt{fi} & \rightarrow F?;A \\
\texttt{if}\, F\, \texttt{then}\, A_1\, \texttt{else}\, A_2\, \texttt{fi} & \rightarrow F_1?;A_1 \cup F_2?;A_2 \\
\texttt{while}\, F\, \texttt{do}\, A\, \texttt{od} & \rightarrow (F?;A)*;(\neg F?)
\end{array}
$$

that provides the precise meaning of the usual structured program constructors. Though the semantic rules of DLProlog can be expressed directly using the structured constructors, they are harder to read and understand.

6 Conclusions

In this paper, the foundations of a PBD system are presented. The PBD system works (i) by learning by demonstration the states and the transitions of the program, represented as predicates and rules, respectively, and (ii) by deriving the entire program by recursively assembling simpler rules. The final outcome of the rule construction process is the program with the intended behavior. This approach may probe to be more realistic than function guessing in applications that may not offer any clue.

Among the future work is to finish the user interface and to develop a solid implementation of the semantic rules of DLProlog in order to axiomatically construct correct programs. The integration of both parts is also a hard task to achieve. However, the aim of this research is encouraged by the teaching experience of the first author in the development and verification of non trivial programs. Building an interactive enviroment that provides the means to teach essential programming concepts is the long-term vision of this work.

References

1. Halbert, Dan: Programming by Example. PhD diss. U.C. Berkeley (1984)
2. Cypher, Allen: Watch What I Do: Programming by Demonstration. MIT Press (1993)
3. Lieberman, Henry: Your Wish is My Command: Programming By Example. Morgan Kaufmann (2001)
4. Aditya, Krishna Menon, et al.: A Machine Learning Framework for Programming by Example. In: Proceedings of the 30th International Conference on Machine Learning, Atlanta Georgia, USA, JMLR: WCP volume 28 (2013)
5. Billard, A., Calinon, S., Dillmann, R., Schaal, S.: Robot Programming by Demonstration. In: Handbook of Robotics, MIT Press (2008)
6. Calinon, S., Guenter, F., Billard, A.: On Learning, Representing, and Generalizing a Task in a Humanoid Robot. In: IEEE Transactions on Systems, Man, and CyberneticsPart B: Cybernetics, Vol. 37, No. 2, pp. 226–298 (2007)
7. Buss, Samuel R.: An introduction to proof theory. In: Samuel R. Buss. Handbook of proof theory. Elsevier, pp. 1-78 (1998)
8. Olmedo-Aguirre, José Oscar, Morales-Luna, G.: A Dynamic-Logic-based Modal Prolog. In: Proceedings MICAI 2012, CPS IEEE Computer Society, pp. 3–9 (2012)