A Direct Manipulation Language for Explaining Algorithms

Jeremy Scott
MIT CSAIL
Cambridge, MA 02139
Email: jscott@csail.mit.edu

Philip J. Guo
MIT CSAIL / University of Rochester
Cambridge, MA 02139
Email: pg@cs.rochester.edu

Randall Davis
MIT CSAIL
Cambridge, MA 02139
Email: davis@csail.mit.edu

Abstract—**Instructors typically explain algorithms in computer science by tracing their behavior, often on blackboards, sometimes with algorithm visualizations. Using blackboards can be tedious because they do not facilitate manipulation of the drawing, while visualizations often operate at the wrong level of abstraction or must be laboriously hand-coded for each algorithm. In response, we present a direct manipulation (DM) language for explaining algorithms by manipulating visualized data structures. The language maps DM gestures onto primitive program behaviors that occur in commonly taught algorithms. We performed an initial evaluation of the DM language on teaching assistants of an undergraduate algorithms class, who found the language easier to use and more helpful for explaining algorithms than a standard drawing application (GIMP).**

I. INTRODUCTION

Instructors in introductory CS courses typically explain algorithms by sketching an example data structure, then carrying out the algorithm's operations on the example. Whether done on the blackboard or in a digital medium, this process can be tedious. Blackboards do not afford manipulation of the drawn objects, requiring the instructor to either erase and redraw the data structure, or storyboard the behavior as a series of before-and-after snapshots. Figure 1, for example, shows an instructor explaining an AVL insertion by drawing the entire binary tree after each step of the rotation.

Digital drawing applications (e.g., PowerPoint, GIMP) allow shapes to be manipulated, but the manipulations are not meaningful in a programming context. For example, numbered boxes might be used to represent an array, but will respond to drag gestures as a set of independent objects, rather than as a list of numbers; as a result, it is not as easy as it should be to demonstrate sorting operations.

An alternative is to prepare animations that illustrate the algorithm's behavior. While these animations are typically rich with programming-specific details they are difficult to create because visualization code (ex. in Processing or JavaScript) must be written for each algorithm; the authoring process is not as direct as drawing and manipulating objects.

In response, we have developed a direct manipulation (DM) language for tracing the operation of algorithms on example data structures. A visual vocabulary of numbers, lists, binary trees and graphs enables the teacher to setup an example data structure. The DM language provides a mapping from gestures to primitive operations, enabling the teacher to illustrate the *concrete trace* of an algorithm by manipulating the example data structure.
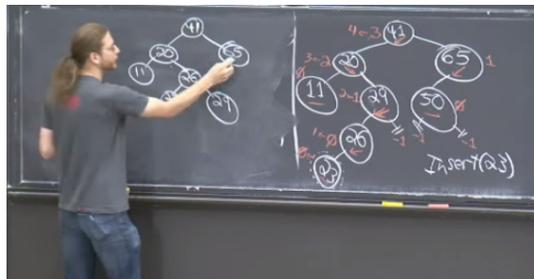

Fig. 1.   An instructor demonstrating an AVL tree insertion.

We performed an evaluation of the DM language's usability and utility by asking teaching assistants from an introductory CS course to trace the behavior of two sorting algorithms on examples of lists. We asked them to compare their experiences using both the DM language and a standard computer drawing application (GIMP) commonly used to describe data structure manipulations in online learning scenarios.

In this paper, we:

1) describe a novel direct manipulation (DM) language for tracing algorithms on example data structures,
2) report on a comparative study of the DM language against a standard digital drawing application for the task of tracing list sorting algorithms on examples.

II. RELATED WORK

Algorithm visualizations are semantically-rich, hand-coded animations of an algorithm's behavior. The fact that they are hand-coded for each algorithm means that the barrier to creating such animations is high. Shaffer et al. surveyed CS instructors and found that while there was high interest in these visualizations, few actually used them in practice due to the difficulty of finding and integrating suitable visualizations into their curriculum [1]. We created our DM language as a means of lowering the barrier to authoring algorithm explanations.

Program visualization tools [2] enable the user to enter a small program, single-step through its execution, and see its visual state at each step. Sorva et al. provide a comprehensive survey of 44 such tools for languages such as Java, C++, and Python [3]. In contrast to algorithm visualization, these tools visualize the low-level semantics of a specific programming language using constructs such as stack frames and pointers. The visual vocabulary and DM language proposed in this work

enable explanations at a higher level of abstraction, in terms of data structures and algorithms.

Visual programming languages enable programmers to write programs via direct manipulation of graphical elements rather than by typing text. These languages have gained widespread adoption in two main areas: (1) domain-specific languages for specialists, such as LabVIEW for electronic systems designers and Max/MSP for digital music creators, and (2) educational programming environments such as Alice [4] and Scratch [5], which allow novices to create simple programs by snapping together colorful blocks. Our DM language is also intended for educational scenarios, but focuses on the CS instructor's typical task of explaining algorithms.

III. DIRECT MANIPULATION LANGUAGE FOR TRACING ALGORITHMS

Our direct manipulation (DM) language maps gestures onto primitive program behaviors that occur in commonly taught algorithms. The design of the language was inspired by watching instructors explain concepts in the introductory algorithms course at our university. We present the design decisions in this language then describe its constituent gestures in detail (Table II).

A. Scope: concrete traces and visual problems

The language's scope was defined in two ways to suit its role in educational scenarios. First, we observed that instructors explain algorithms by drawing concrete examples of data structures - for example, a specific graph when explaining Dijkstra's shortest path algorithm - and then trace the algorithm's behavior on that example. We call this a *concrete trace*, because the instructor does not illustrate the algorithm in the abstract, and instead carries out its execution as a sequence of concrete steps. For this reason, the DM language focuses on concrete changes to data structures, rather than flow control diagrams of loops or conditional branches.
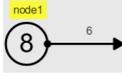
Second, our language focuses on supporting algorithm traces for lists, binary trees, and graphs, because their canonical algorithms – sorting, rotation, search and shortest paths – are widely taught in introductory CS and algorithms courses. These data structures and their associated algorithms are also inherently visual: the natural way to teach and think through them is by drawing diagrams. Thus, we envision direct manipulation languages for describing computation to be most useful for these types of problems.

B. Visual vocabulary

Instructors use a relatively consistent visual vocabulary to draw data structures: lists as rows of boxes, and trees and graphs drawn as circles connected by lines or arrows. Typically, the primitive part of a data structure (single list element, node or edge) takes on a numerical value, which is sketched inside the box or circle, and adjacent to the arrow. Our DM language portrays data structures using this common visual vocabulary (Table I).

Our observations of instructors led us to expand this vocabulary. We noted that when describing insertion sort, instructors indicated the current element to be inserted into

TABLE I. VISUAL VOCABULARY OF DATA STRUCTURES

Data Structure	Visualization
Number	
List	
Binary Tree Node	
Graph Node	
Graph Edge	
Finger	

the sorted sublist by drawing an arrow that points to it. We therefore included a *finger* visualization in the language, which can be used to keep track of where we "are" as the algorithm progresses.

We also noted that instructors verbally explain decision making when they are illustrating a concrete trace of the algorithm. For example, during an AVL insertion, the instructor might say: "because the 6-node is greater than the 4-node, we follow the 4-node's right pointer." For this reason, we added numerical comparisons and pointer traversals to the visual vocabulary. These are intended to help illustrate basic decision making during the tracing task.
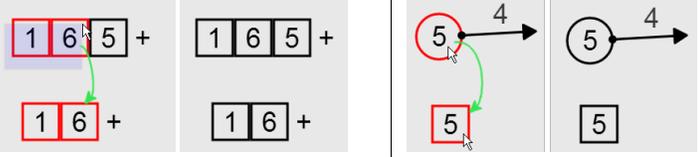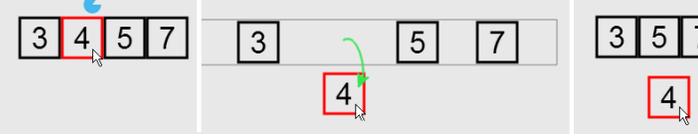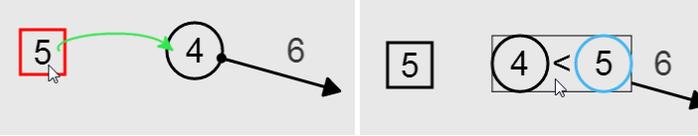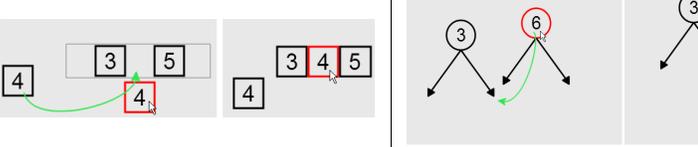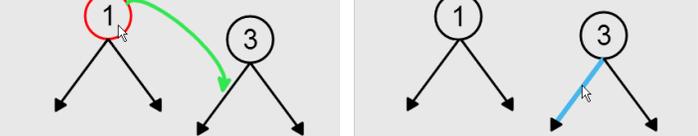
C. Gestural vocabulary

To design this vocabulary of DM gestures, we examined a variety of examples of sorting and search algorithms being explained by instructors and asked three questions:

- What behaviors need to be expressed to explain each algorithm's trace on the example data structure?

- How do instructors currently express those behaviors by drawing on the blackboard?

- What direct manipulation gestures would be most natural to express those behaviors?

For example, insertion sort requires the instructor to express atomic program behaviors such as: create a list, maintain a pointer to the current list position, compare numeric values, and rearrange elements. AVL insertions are expressed by creating binary tree nodes, comparing node values, recursing into a node's left or right child, inserting nodes, and rotating subtrees. Graph algorithms involve similar but unconstrained constructions of nodes and edges, in addition to comparisons, pointer traversals and updates of node and edge values. For all data structures, instructors may want to mark nodes or list elements to indicate some state (e.g. visited or sorted).

TABLE II. DIRECT MANIPULATION LANGUAGE FOR TRACING ALGORITHMS

Atomic behavior	Direct manipulation	Example(s)
Use or copy an object's value	*drag-away*: Grab an object, and drag it away quickly. Drop it to create a new number with the same value.	
Remove an object from its parent	*dwell-drag-away*: Grab an object, dwell for one second and drag it elsewhere.	
Compare two objects' values	*drag-into*: Drag one value into another.	
Assign one object's value to another	*drag-into-dwell*: Drag one value into another, and dwell until the desired interpretation (=, +=, -=) appears.	
Insert a value into list or node into tree/graph	*drag-insert*: Drag the value into a gap in the list, or the node to the tip of a pointer/edge.	
Show a pointer traversal	*drag-follow*: Drag a value over a binary tree pointer or graph edge.	
Indicate current point in traversal	*drag-finger*: Drag the finger from one element to another.	

Based on this list of behaviors, we devised a set of 8 gestures (Table II) that is both expressive and compact; analogous behaviors can be performed in the same way regardless of their data structure type. For example, dragging one object into another triggers a comparison of their numeric values, regardless of whether they are list elements, nodes or numbers. Similarly, popping a list element or detaching a subtree from its parent is accomplished by grabbing it and holding it – *dwelling* – until it can be moved away freely. This expressivity means the language can be used to describe a variety of list, tree and graph algorithms.

These gestures have been designed in keeping with principles of direct manipulation, both by Shneiderman's original definition [6] and what has been called *post-DM* [7]. Objects in our visual vocabulary act like physical objects with constraints and affordances that match the way programmers think of abstract data structures. For example, when a number is brought into the vicinity of a list, the list expands to offer spaces between existing elements where the number can be inserted. When the number is inserted, the list collapses back into place, in its new configuration.

D. Resolving overloaded gestures

When designing a gesture vocabulary, the gesture that seems most natural for a task can often be interpreted in multiple ways. There are two main examples of this in the DM language: ambiguity in the *drag-into* gesture, and ambiguity in the *drag-away* gesture when the value is part of some higher-level data structure (e.g. a list element, or a child node in a tree or graph).

The drag-into gesture is ambiguous because dragging one object into another could be interpreted as a comparison (x<y), an assignment (x=y), or an augmented assignment such as x+=y or x-=y. Performing the drag-away gesture on a child object is ambiguous because it is not obvious whether the parent object should be altered or not. For example, when

dragging an element away from a list, should a copy of the element be made, or should the element be pulled out of the parent list? For trees and graphs, should a copy of the dragged node be made, or should that node and its descendants be detached from the parent?

Our solution is to default to the gesture interpretation that does not alter any involved data structures. If the user dwells for more than one second, then we preview possible mutation behaviors. For the drag-into gesture this means defaulting to a comparison, then previewing assignments. For the drag-away gesture, a copy of the dragged value is made, unless the user has dwelled and removed the value from its parent data structure.

IV. Evaluation

To evaluate the direct manipulation language, we conducted a comparative study against a standard drawing application (GIMP). We chose GIMP because it is the state-of-the-art in authoring digital explanations of algorithms. In order to test the DM language, we implemented it in a prototype system called *CodeInk*, which will be presented in more detail in future work. The system embodies the described visual vocabulary and DM language, making it possible for subjects to setup example data structures and manipulate them to trace an algorithm.

Our subjects were four teaching assistants (mean age = 23.5, σ=4.0) from the introductory algorithms course at our university. By asking them to compare their experiences using each tool to explain list sorting algorithms, we tested the following hypotheses:

H1: TAs find it easier to explain list sorting algorithms using the DM language than using the drawing application.

H2: TAs find the DM language more helpful than the drawing application for explaining an algorithm's trace.

A. Tasks

Our study had 2x2 conditions: two list sorting algorithms (insertion and merge sort) and two ways of explaining each algorithm (the DM language and the GIMP digital drawing application used with a Wacom pen tablet). We used a within-subjects design, where subjects explained the two sorting algorithms in both CodeInk and in GIMP. The ordering of tool-algorithm pairs was counterbalanced between subjects to avoid confounding effects.

After subjects watched a training video and had time to become familiar with the DM language, they were asked to explain each algorithm's trace on an example list using both tools. At the end of the study, subjects were asked to fill out a questionnaire rating agreement with two statements: (a) the DM language was easier to use and (b) was more helpful for explaining algorithms than GIMP.

B. Results and Discussion

Since there were only 4 subjects in this evaluation of the language, our results cannot be considered statistically significant. Across subjects, agreement with the 'easier-to-use' statement had a mean score of 5.00 and agreement with the 'more helpful' statement had a mean score of 6.50. Multiple

subjects commented that the affordances and constraints of the DM language "felt realistic for explaining the algorithm," and that "actually swapping the elements was more illustrative than redrawing the list over and over again."

While subjects were able to explain insertion sort much faster in using the DM language, the same was not true for merge sort. For example, when merging one element at a time into the final list, subjects would often accidentally dwell and remove an element instead of copying it. While subjects found comparisons illustrative, they would also sometimes accidentally compare values when dragging a list element across the canvas. Replacing the dwelling construct with contextual menus or more explicit controls for entering `copy`, `remove`, `compare` or `assign` modes might resolve some of these usability problems.

V. Conclusion

This paper presents the design of a direct manipulation (DM) language for describing algorithm traces on examples of data structures. The language provides a mapping from DM gestures to primitive program behaviors that occur in list, binary tree, and graph algorithms commonly taught in CS courses. In a comparative study that evaluated the language's usability and usefulness against a standard drawing application, teaching assistants in an introductory algorithms course found CodeInk easier to use and more helpful for explaining list sorting algorithms. We have implemented the DM language in a system called CodeInk; its design and evaluation will be described in future work.

References

[1] C. A. Shaffer, M. Akbar, A. J. D. Alon, M. Stewart, and S. H. Edwards, "Getting algorithm visualizations into the classroom," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 129–134.

[2] P. J. Guo, "Online Python Tutor: Embeddable Web-based Program Visualization for CS Education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 579–584.

[3] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *Trans. Computing Education*, vol. 13, no. 4, p. 15:1–15:64, Nov. 2013.

[4] W. P. Dann, S. Cooper, and R. Pausch, *Learning To Program with Alice*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1477661

[5] J. L. Ford, *Scratch Programming for Teens*, 1st ed. Boston, MA, United States: Course Technology Press, 2008.

[6] B. Shneiderman, "The future of interactive systems and the emergence of direct manipulation," *Behaviour & Information Technology*, vol. 1, no. 3, pp. 237–256, 1982.

[7] B. Lee, P. Isenberg, N. H. Riche, and S. Carpendale, "Beyond mouse and keyboard: Expanding design considerations for information visualization interactions," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 12, pp. 2689–2698, 2012.