Adapting Higher-order List Operators for Blocks Programming

Soojin Kim and Franklyn Turbak
Computer Science Department
Wellesley College
Wellesley, Massachusetts, USA
Email: {skim22,fturbak}@wellesley.edu

Abstract—In MIT App Inventor (AI), puzzle-shaped blocks are connected to program Android apps. AI has Python-like lists typically manipulated with loops, but implementing such loops correctly is challenging for novice AI programmers. To simplify list processing, we extended AI with new blocks that map, filter, reduce and sort lists. Since AI does not have first-class functions, these pseudo-higher-order list operator (PHOLO) blocks incorporate the parameter and body declarations of functional arguments normally associated with these operators. To assess the usability of these new list operators, we conducted a user study with 18 students with AI experience. Most users correctly completed a majority of the tasks, but many struggled with processing lists of lists and sorting tasks involving two keys.

I. INTRODUCTION

Blocks programming languages, in which program syntax trees are represented as compositions of visual puzzle-shaped blocks, are increasingly used in introductory programming environments. Tens of millions of people of all ages and backgrounds have programmed in blocks-based environments, which include Scratch ([1], [2]), Blockly [3], Snap! [4], Pencil Code [5], and Code.org’s Hour of Code exercises [6]. Our focus in this paper is MIT App Inventor (AI) [7], in which a blocks language is used to specify the behavior of components in a mobile app for Android devices.

AI’s only data structure is a Python-like list, an indexed sequence of mutable slots, each of which contains an item. AI lists are often processed using loops that iterate over the items or indices of the list. There are three looping blocks in AI: a for each item loop that iterates over each item in a list (see Fig. 1); a for each number loop that iterates over a range of numbers; and a while loop that iterates until its test expression becomes false.

In our experience, AI beginners frequently have trouble correctly processing lists with loops. For example, in a CS0 mobile apps course taught by Turbak using AI in Fall 2014 [8], 9 of 20 apps for the final two projects had at least one incorrectly defined loop. All 9 failed to properly initialize a global variable in at least one loop. In the context of Fig. 1, this bug would be failing to include the block that sets the global variable filteredPhrases to the empty list at the beginning of the Click event handler. Because this global variable is initialized to the empty list when it is declared at top level, the handler will work the first time the button is clicked, but not subsequent times. There were other loop troubles as well, such as index-out-of-bounds errors, and unnecessarily complex loops that used a for each number loop to select list items by their indices instead of the simpler for each item loop to iterate over the items directly.

Can such problems be mitigated by discouraging the use of global variables? AI programs can sometimes be restructured to replace global variables by local ones, but the event-based nature of AI programming forces many variables to be global [9]. In the case of Fig. 1, if filteredPhrases is referenced by other event handlers in the game, it must remain global.

Many AI blocks encapsulate high-level, easy-to-use abstractions that hide many low-level complexities. Keeping with this philosophy, we decided to address common problems with loop-based list processing by providing new high-level list-processing blocks (map, filter, reduce, and sort) that capture common looping patterns. These were inspired by similar list operators in functional languages (e.g., Scheme/Racket, Haskell, ML), as well as in Python and JavaScript.

Functions like map, filter, reduce, and sort are known as higher-order list operators (which we shall abbreviate as HOLOs) because they take functions as arguments. Since AI does not currently support first-class function values, HOLOs cannot be directly added to AI. We solve this problem by creating specialized blocks that we call pseudo-higher-order list operators (PHOLOs) that have the parameters of the usual functional arguments baked into the blocks. For example, Fig. 2 shows how the Click handler from Fig. 1 can be simplified with our new filter block, which, in addition to a list argument, has an item parameter that ranges over the list items and a boolean test expression that determines which items are kept in the new list returned by the block. Moreover, by avoiding directly initializing and appending to the global filteredPhrases list and instead updating it after the filtering process, we completely avoid the common pre-loop global variable initialization bug described above.

In the rest of this paper, we present the design of our PHOLO blocks and describe a study we conducted to evaluate their usability. Our contributions are (1) developing a way to express powerful HOLO-like operators using PHOLOs in a blocks language that does not support first-class functions and (2) assessing the usability of these PHOLO blocks.

II. DESIGN OF PSEUDO-HIGHER-ORDER LIST BLOCKS

A. Map, Filter, and Reduce Blocks

Fig. 3 displays our PHOLO map, filter, and reduce blocks. Following AI’s convention, verbose labels convey the meaning of blocks and their sockets. So what we call the map block is actually labeled make new list from input list socket mapping each item parameter to indented body socket. Rather than taking a function as an argument (which is
Fig. 1. The Click handler initializes global variable filteredPhrases to an empty list, and then the for each item loop populates it by all strings in the global list phrases (collapsed to hide details) whose string length is at least the number specified in the MinPhraseLength text box. filteredPhrases is then used in a call to the updateGameInterface procedure (whose definition is not shown). As the loop executes, the loop variable item successively takes on the value of each list item.

Fig. 2. The Click handler with our new filter block (labeled make new filtered list from). It declares a baked-in item parameter that ranges over all input list items and is referenced from the test expression.

impossible in AI, these blocks incorporate both (1) the parameter(s) of that function (as the salmon-colored parameter(s) on the block) and (2) a hole for the body of that function (the indented expression socket below the parameter). Together, these two parts play the same role as the usual functional argument for the traditional HOLO versions of these blocks. This relationship is clarified in Fig. 4, which shows a Python map function called on an anonymous lambda incrementing function and a list of numbers and how this is expressed with the AI map block. Note how the item parameter of lambda is incorporated into the map block, and the body of the lambda is provided as a body expression (in the scope of item) that fills the indented expression socket of the map block.

Fig. 3. The PHOLO map, filter, and reduce blocks for AI.

Fig. 4. The anonymous function’s item parameter is baked into the map block, and its body fills map’s expression socket in the scope of this parameter.

Fig. 5. Dolt bubbles show the dynamic behavior of the map example.

Fig. 6. A reduce block that sums the lengths of strings in its input list.

B. Sort Blocks

AI currently does not have a list sorting operator. One can be defined as an AI procedure, but this is beyond the capabilities of most beginning AI programmers. We fix this problem by providing three sort blocks (Fig. 7). The first, sortbasic, has no baked-in parameters and uses a default comparator. It returns a new list that is a sorted version of the input list, where elements are sorted in ascending order first by type and then by values within each type, using an arbitrary ordering on types and standard orderings on values.

Fig. 7. The sortbasic, sortkey, and sortcomp blocks.

The second sort block, which we call sortkey, has a single item parameter and a socket for a body expression. Together, these determine a proxy value that is used to sort the item by the default comparator of sortbasic. E.g., in Fig. 8, sortkey sorts a list of strings in ascending order by their lengths.

Fig. 8. Using sortkey to sort strings by their lengths. The sort is stable: items with equal proxy values keep their same relative order from the input list to the output list. This is why Zoe appears before Sam in the output list.

The third sort block, sortcomp, has two parameters, item1 and item2, and a socket for a boolean body expression. Together, these denote a less-than-or-equal-to comparison function. If the body expression evaluates to true, then item1 will precede item2 in the output list; otherwise item2 will precede item1. For example, in Fig. 9, the comparator specifies that the strings should be sorted in descending lexicographic order.

be explained in the context of Fig. 2. The reduce block (Fig. 6) combines the items of the input list into an answer value. It begins with the input initialAnswer (the answer when the list is empty; 0 in this example), and processes the items left-to-right, combining the current answer (named by answerSoFar) with the current list item (named by item) to produce the next answer. The combination is performed by the body expression. In the example, the input list has four strings, and the output is the sum of the lengths of these strings.
III. USER STUDY

A. Purpose and Structure of the Study

To assess the usability of PHOLO blocks, Kim facilitated a user study with 18 Wellesley College students familiar with AI from a course or a project. The purpose of this study was to determine if the new list operators are usable by AI’s target audience — users with limited programming background. Here we summarize key aspects of the study and its results; for details, see Kim’s undergraduate dissertation [10].

Each user study lasted up to 90 minutes. Participants first filled out a pre-task survey indicating age, major(s), previous knowledge of AI and HOLOs (if any), and CS courses taken. All were women aged 18–23. They had 7 different majors, with 11 users majoring in CS or double majoring in CS and another subject. To control for skill level and background, we divided them into two groups: Group 1, the 10 students (56%) who had previously seen HOLOs in Python, OCaml, or Scheme/Racket; and Group 2, the 8 students (44%) without such exposure.

The next step was a brief tutorial on each PHOLO block, followed by a set of tasks that involved writing new programs and explaining the meaning of programs that use the PHOLO blocks. Kim took notes during each study and recorded a screencast capturing the user’s voice and actions on the screen.

The first part of the study consisted of eight tasks involving mapping, filtering, and/or reducing over a simple list or a list of lists. The second part of the study involved six sorting tasks on a list of lists. Users tested their programs by connecting the computer to an Android device and running Dolt on the blocks. The study ended with a post-task survey where users indicated why it was easy or difficult to use the PHOLO blocks and shared any suggestions for improvements.

B. Results for Mapping, Filtering, and Reducing Tasks

Fig. 10 shows the successful task completion rate for the 8 tasks involving map, filter, and reduce. Group 1 had an average success rate of 98% on these tasks, compared to 67% for Group 2. All Group 1 users correctly completed at least 7 of the 8 tasks. Two could not explain the meaning of a program that filtered, mapped and then reduced over a list of lists. But all Group 1 users correctly completed the synthesis task using a combination of these blocks. In Group 2, all users successfully mapped and filtered over a simple list. But many struggled with the concept of reduction, with using any of these operators on a list of lists, and with using these operators in combination with one another.

Something not reflected in Fig. 10 is that many users who eventually correctly completed a task encountered some problems along the way. They often referred back to tutorial examples to figure out how to use the blocks correctly, especially when they encountered an error. Overall, students in

C. Results for Sorting Tasks

For the six sorting tasks, Fig. 11 shows that Group 1 again performed just as well or better than Group 2 on all the tasks.

Overall, the results show that a large majority of users in both groups were able to sort the given list in ascending or descending order by the element itself or by one key. However, sorting with two keys proved particularly challenging. This was in part due to flaws in the study involving unexplained subtleties of the stability of the sorting algorithm as well as the particular list being sorted; see [10] for details.

There were two common problems encountered during the sorting tasks. First, when using sortcomp, a total of nine users
(50%) tried to use the boolean test item1 > item2 to sort the list in increasing order or the test item1 < item2 to sort the list in decreasing order. Six users (60%) in Group 1 and three users (38%) in Group 2 made this mistake. We thought that the English phrasing "where item1 precedes item2 if" on the sort block would strongly suggest the correct interpretation, but the results say otherwise. The second confusion was experienced by three users in each of Group 1 and Group 2: trying to use sort (which sorts in increasing order only) for tasks that required sorting in decreasing order. These problems indicate that the sort blocks need to be better explained in documentation and tutorials.

In the post-task survey, most users indicated that they found sort basic and sort comp intuitive to use. Several, however, said they did not understand how and when to use sort key and asked for more examples involving keys. Others did not like having to choose between sort key and sort comp. Since sort comp can do everything that sort key does (and more), we plan to eliminate sort key going forward.

IV. RELATED WORK

Like AI, several other blocks languages with list or array-like data structures, including Scratch [1] and StarLogo Nova [11], do not have HOLOs, so users use loops with list indexing to iterate through list elements. A notable exception is Snap! [4], a blocks language based on Scratch that supports Scheme-like higher-order functions (HOFs), including list mapping, filtering, and reducing blocks. In the Snap! mapping example in Fig. 12, the multiplication block with an empty input surrounded by a gray ring denotes a one-arg doubling function. The gray ring wrapper is a concise visual notation for declaring an anonymous function [12].

The Droplet editor ([13], [14]) for Pencil Code [5] and Code.org’s App Lab supports isomorphic conversions between the textual notations of existing languages and a blocks notation. When it is used for languages with anonymous first-class function declarations, like dialects of JavaScript, the shapes of the resulting function blocks resemble gerrymandered districts (Fig. 13). Just as AI avoids the complexity of first-class function callbacks in programs like Fig. 13 with its top-level parameterized event handlers [9], PHOLO blocks similarly avoid the complexity of blocks for functional arguments.

Numerous older visual languages support higher-order operators on lists and arrays. Most of these are dataflow languages. In Show and Tell [17], iteration boxes with so-called parallel ports allow processing of individual array elements. LabView’s tunneling and auto-indexing features similarly allow processing array elements in loops [18]. Both Extended Show And Tell [19] and DataVis [20] extend Show and Tell with means of passing and applying HOFs. Prograph’s list annotations enable specifying code that is run on each element of a list, and its inject feature allows specifying a function argument by name [21]. VPL [22] and VisaVis [23] are dataflow systems for functional programming with an emphasis on HOFs. Tonic [24] uses a dataflow visualization with support for HOFs to specify reactive web-based multi-user applications. One non-dataflow language in this category is CUBE ([25], [26]) a three-dimensional visual logic programming language that can express HOLOs like filter. We know of no studies evaluating the usability of HOLO features in these languages.

V. CONCLUSION AND FUTURE WORK

We have implemented our PHOLO blocks in an experimental version of AI, and it is being reviewed for integration into the master version. Although beginners can use them in some situations, our user study indicates that more work needs to be done to make these operators less confusing to those who have not previously seen similar operators. In addition to improving documentation and tutorials, we need to carefully finalize the labels and parameter names on PHOLO blocks. We have recently learned about the syntactic choices made in the evidence-based programming language Quorum ([27]) and wonder if a similar methodology could be applied here.

In our user study, we only tested the usability of PHOLO blocks and did not test whether list processing with these blocks is easier or harder than with loops. We plan to conduct a user study to investigate this, and aim for this study to have a wider demographic base (in terms of age, gender, and background) than the one reported here.

ACKNOWLEDGMENTS

This work was supported by Wellesley College Faculty Grants, by sabbatical funding from Wellesley College, and by the National Science Foundation under grant DUE-1226216.
REFERENCES