A Tangible Interface for Learning Recursion and Functional Programming

Juan Diego Tascón Vidarte
Department of Advanced Technology Fusion
Konkuk University
Seoul, Republic of Korea
jtascon@konkuk.ac.kr

Christian Rinderknecht
Department of Internet and Multimedia Engineering
Konkuk University
Seoul, Republic of Korea
rinderkn@konkuk.ac.kr

Jee-In Kim
Department of Internet and Multimedia Engineering
Konkuk University
Seoul, Republic of Korea
jnkm@konkuk.ac.kr

HyungSeok Kim
Department of Internet and Multimedia Engineering
Konkuk University
Seoul, Republic of Korea
hyuskim@konkuk.ac.kr

***Abstract*—Recursion is a powerful programming technique which is notoriously difficult to master, especially in functional languages because they prominently feature structural recursion as the main control-flow mechanism. We propose several hypotheses to understand the issue and put some to the test by designing an open-source interactive interface based on a tangible block-world with augmented reality and software feedback. Stacks of blocks are used as an analogy for the list data structure, which enables the simplest form of structural recursion. After using this application, students are expected to transfer their training to directly write recursive programs in sequential Erlang, a purely functional language.**

***Keywords*-functional programming, tangible user interface, block world, augmented reality, software feedback**

I. Introduction

Recursion is the syntactic property of a definition to be self-referential. In high-school mathematics, recursion is commonly found in the guise of *progressions* and *proofs by induction*. Nevertheless, when college students are introduced to recursion both as a design and a programming method, they show great difficulties to understand it. Some researchers have been trying to tackle the issue through the lenses of *cognitive sciences*, inferring the mental models of recursion [1], [2], in particular the faulty ones that students construct by interacting with the teacher and the problem to solve. Often, these models posit a *constructivist theory of learning*, [3], [4] where it is assumed that the learners build their knowledge based upon previous idiosyncratic conceptions. Within this framework, the self-referential nature of recursive definitions might seem a priori a challenge; moreover, the curriculum followed by the learner must be analysed and compounded into the analysis.

Some researchers insist more on the first aspect and bring to the fore the essentially linguistic nature of the didactical relationship between learners and teachers in a traditional classroom. They set up experiments, record all the interactions, sometimes including video, and analyse the transcripts to pinpoint the misunderstandings and trace them back to plausible causes [5], [6], [7], [8].

Some other researchers focus more on the curricular aspect and note that most students are first taught *iteration*, which presupposes loops and side-effects, and that their model of execution of the programs is fixed early, thus hampering their later grasping of recursion which relies solely on function calls and perhaps not even on imperative features. Recursion is then thought as a kind of iteration, leading to errors, especially when functions are not tail-recursive, that is, when the flow of control is bidirectional [9].

Recursion is best conceptualised in a declarative model of the machine, allowing an easier analysis of the problem and bridging the gap with a program solving it, often called the *Divide and Conquer* method. This approach becomes the main feature in *purely functional languages*, which rely on mathematical functions and immutable data, so recursion is the sole control-flow mechanism.

But forcing recursive design upon students may not come without some drawbacks [10], [11], therefore a more conservative option consists, within a course on procedural or object-oriented programming, to teach structural recursion on singly-linked lists before arrays and loops [12], or to rely on mathematical exercises dovetailing high-school courses [13], or to use fractal geometry as an intuitive support. The latter alternative resorts to the area of algorithm visualization and animation, [14] which has lead to the proposal of multimedia tools [15], visual examples [16], [17] and programming environments [18] to support teaching and program development. As a contrast, little work has been done on the *transfer of training* from visual interfaces to textual programming, an approach in which the interface is only a temporary tool [19].

Our proposal lies at the confluence of these different streams of investigation. We are interested in exploring the design of interfaces other than the commonplace monitors, mice and touchpads, and using them as an aid to learn recursive programming. More precisely, we designed and implemented a *tangible user interface* (TUI). In order to provide *software feedback* to the learner, we complemented the interface with *augmented reality* (AR). The target language to which transfer of training skills is expected is the sequential subset of Erlang, which is purely functional. This election was motivated by the extremely simple syntax of this language. We only aim at teaching structural recursion

on lists by means of a single tail-recursive function and that is why a *block-world* analogy sustains the TUI: a list in Erlang is what is called a *stack* in algorithms, so we rely upon a stack of physical blocks to represent it. Because of gravity, the interaction of the novice with the blocks is then naturally guided to the manipulation of the topmost blocks, paralleling the two basic Erlang operations of *popping*, i.e., lifting a top block, and *pushing*, i.e., putting a block on top. Manipulating another block would lead the whole stack to collapse. Tail-recursion allows us to ignore all concerns about representing the *control stack*, which holds instances of the call contexts.

II. Methodology

System overview. The application challenges the student to solve a problem by interacting with stacks of blocks placed on a table. A complete session is best understood as a series of scenes, that is, snapshots of the board, which abstracts the execution trace of an Erlang function on a given input, i.e., the initial setting of the board. Markers are signs that are captured and recognized by the system from the physical scene. They are specifically designed to be easily understood by the ARToolKit (which is a back-end for OSGART) and are stuck on each block. They come in three kinds: *Item*, *Stack* and *Switch* (whose occultation by the hand triggers a snapshot). For each recognized marker, a number, a variable, a picture may be superimposed to the video feed and sent to a monitor. This 3D-capable AR feature is supported by another back-end of OSGART called OpenSceneGraph.

Interactions. The stack data structure can be modified only by two simple operations: pop and push. These are the minimum set of operations required to allow us to change the data in any way we wish. However, in our system it is necessary to recognize the block source and destination, that is, a stack, the table (it then represents a value which is not a stack) or outside/inside the scene, so the resulting set of operations is the following:

- *Pop:* moving the top block from a stack onto the table;
- *Push*: moving a block from the table onto the top of a stack (dual of *Pop*);
- *Pop-Push:* moving a top block from a stack on top of another stack (logical composition of *Pop* and *Push*);
- *Discard:* moving a block or stack out of the scene, regardless of where it is located;
- *Create:* moving a block or a stack from outside the scene onto the table or the top of a stack (dual of *Discard*).

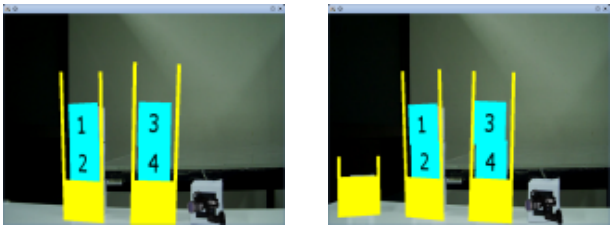
Changing a scene by any other means or by combining at once two or more of these operations is considered invalid. These situations are detected by the application which then stops the session with an error message explaining what happened and prompts the student to restart from scratch by placing the blocks back into a valid initial scene.

The system offers two execution modes: one called *free*, allowing the student to freely, but validly, move blocks and another one called *supervised* in which each operation is checked to conform to a possible solution, otherwise it is rejected and signalled by a visual cue.

Adding new problems requires validating initial and final scenes and dynamically creating *positive scenarios*, that is, admissible execution traces for different possible Erlang functions corresponding to the problem. There are currently five problems defined: stack reversal, joining two stacks, removing all the occurrences of a given block in a stack, removing all consecutively repeated blocks and sorting by insertion. The last two exercises are more than structurally recursive because their assume some property on the denotations of the blocks, equality and a total order, respectively.

A session. Through the interface, blocks augmented in yellow denote the base of stacks and light blue blocks represent items belonging to a stack. The block with a camera icon projected upon stands for the switch that, when hidden, triggers a scene snapshot. A session ends with the system superimposing a congratulation message if the student provides a correct and validly obtained answer. In the case of an invalid action, the system displays a message indicating the cause of the mistake, the blocks involved in the invalid move will be augmented with red and the student will have to restart the session from the beginning.

To illustrate how the interface works, we are going to step through a sample session of the exercise consisting in joining two stacks, that is, forming a new stack by moving all the items of one stack onto the top of another, while retaining their original order. Textually, in Erlang, this means that, given the stacks `P` and `Q`, with `P` denoting `[1,2]` (the top is `1` and the bottom `2`) and `Q` denoting `[3,4]`, the joining of `P` and `Q` is `[1,2,3,4]`. The user starts the session by setting two stacks on the table, representing the input arguments `P` (on the left) and `Q` (on the right) in Erlang, i.e., `join(P,Q)` is the function call to compute. Some random integers are superimposed on the blocks (they are not random in the coming snapshots, so it is easier to follow the transformations). This scene can be seen in Figure 1a. The change of scene is denoted in Erlang by an arrow `->`



(a) `join([1,2],[3,4])` (b) `join([],[1,2],[3,4])`

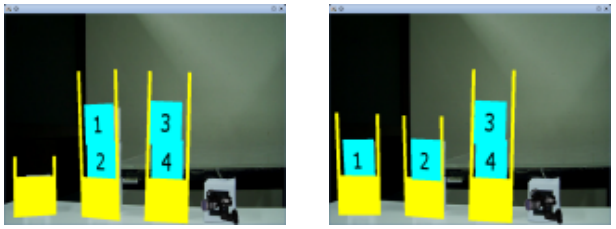
Figure 1: `join(P,Q) -> join([],P,Q)`

at the left of which is a pattern for the current scene and at the right is the next scene in terms of the previous one. The final result `[1,2,3,4]` is shown on the display and the user is prompted to introduce from outside the scene an

empty stack at the leftmost side of the scene, which will be used as a temporary accumulator (operation *Create*). This new scene is in Figure 1b and the Erlang piece of code whose instanciation applies here is

```
join(P,Q) -> join([],P,Q).
```

where `[]` represents the empty stack. The user is now expected to move top blocks from the middle stack (`P`) to the left stack (operation *Pop-Push*), as shown in Figure 2. This transformation is captured by the Erlang clause

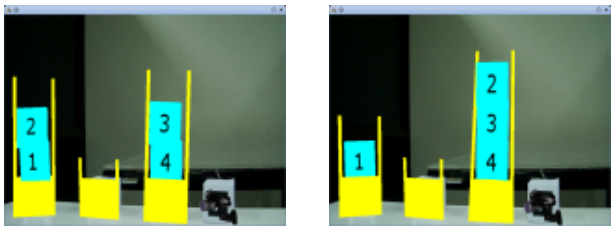


(a) `join([],[1,2],[3,4])` (b) `join([1],[2],[3,4])`

Figure 2: `join(A,[I|P],Q) -> join([I|A],P,Q)`

```
join(A,[I|P],Q) -> join([I|A],P,Q);
```

In Erlang, `[I|P]` is a pattern which matches any non-empty stack whose top block is named `I` and the remaining stack `P`. After repeating this operation one more time on our example, the scene is as shown in Figure 3a. It is matched by



(a) `join([2,1],[],[3,4])` (b) `join([1],[],[2,3,4])`

Figure 3: `join([I|A],[],Q) -> join(A,[],[I|Q])`

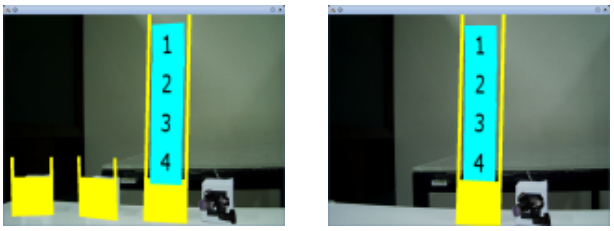
the pattern `join([I|A],[],Q)`, meaning "left stack with top block `I` and sub-stack `A`, middle stack empty and right stack `Q` unchanged." The next phase consists in moving the blocks from the leftmost stack to the rightmost (operation *Pop-Push*). This is shown in Figure 3, which corresponds to an instance of the piece of source code

```
join([I|A],[],Q) -> join(A,[],[I|Q]);
```

After repeating this operation once more on our example, the left stack becomes empty and the corresponding scene is seen in Figure 4a. The Erlang pattern for it is `join([],[],Q)`. The result is finally reached by keeping only the non-empty stack on the scene (operation *Discard* twice), as shown in Figure 4. This is expressed by the clause

```
join([],[],Q) -> Q.
```

The student is not presented the Erlang at this stage, we wanted to show the analogy between the AR scene and the textual program, which the student in a later session will be asked to write directly. Here, the complete Erlang program of this session was as follows (module declarations omitted):



(a) `join([],[],[1,2,3,4])` (b) `[2,3,4]`

Figure 4: `join([],[],Q) -> Q`

```
join(          P,Q) -> join(   [], P,    Q).
join(    A,[I|P],Q) -> join([I|A], P,    Q);
join([I|A],   [],Q) -> join(    A,[],[I|Q]);
join(   [],   [],Q) -> Q.
```

Scope and limitations. From the standpoint of programming expressivity, the system is limited to functions based on structural recursion on stacks and constant values. This is a consequence of using a TUI, so adding new interactions would require technologies that are not widely spread, for example gesture recognition, motion tracking, voice recognition, etc. On the other hand, with the current system, it just as easy to use rectangular pieces of paper on a table instead of blocks, making it as portable as the laptop with a webcam running it. The set of definable function is also restricted to tail-recursivity because representing the control stack would require a special stack growing top-down, so the order of the instances of the control contexts is preserved. Since the system is intended to be used by novices as a temporary tool to understand simple cases of recursion, instead of as a general programming environment, this is not an impediment.

Future experiments. The first phase is made up of three stages, which are repeated. In a first stage, the students will be asked to build some stacks and randomly chosen integers will be displayed on the blocks, one per block. The expected result will in turn be shown on the screen. The student may select either the free or supervised execution mode and try to reach the goal. After, two more random examples for the same problem will be proposed. This stage relies only on *concrete examples*, as the one illustrated above with `join`. It presents some similarities with the framework of *programming by example*, except that the goal is not to teach the computer how to program but the other way round.

In the second stage, another random example for the same problem is displayed but only the top of the stacks will be visible. This is meant to induce in the learner the understanding that the exercise can be solved without relying on global knowledge and that *information hiding* actually helps to focus the attention on the smallest part of the data which is needed to make one more little step towards the result. Each time a *Push-Pop* operation is performed, for instance, only the topmost blocks are consistently shown.

After three runs like this, three other examples of the same problem are offered, but instead of integers superimposed on the block markers, *variables* are. This last stage aims at giving rise to *abstraction* and prepares the transfer to textual programming.

When the last stage is over, another problem is submitted to the student etc. until the interface hopefully becomes useless and the direct programming of the Erlang functions corresponding to the previous exercises is attempted. In the second phase, the professor shows the analogy between, on the one hand, the stacks of blocks and the valid operations learnt by means of the interface and, on the other hand, the Erlang syntax for lists and the Erlang semantics. We expect to measure a statistically significant transfer of training for students who used the system in comparison with students who did not.

III. Conclusion

We explored a tangible interactive tool that will be used to help beginners to learn how to recursively solve simple problems on stacks of blocks and then how to transfer their training to the writing of the corresponding Erlang functions.

Many technical efforts have been spent in order have the application run on GNU/Linux and be licensed as open source, so the interested researchers may benefit from the freedom to study it, use it, modify it and redistribute it under the same terms.

References

[1] I. Sanders, V. Galpin, and T. Götschi, "Mental models of recursion revisited," in *Conf. on Innovation and Technology in Comp. Sci. Education*. Bologna, Italy: ACM SIGCSE, Jun. 2006, pp. 138–142, ISBN 1-59593-055-8.

[2] C. Mirolo, "Mental models of recursive computations vs. recursive analysis in the problem domain," in *Conf. on Innovation and Technology in Comp. Sci. Education*. Paris, France: ACM SIGCSE, Jul. 2009, pp. 397–397, ISBN 978-1-60558-381-5.

[3] M. Ben-Ari, "Constructivism in computer science education," *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, Jan. 2001, ISSN 0731-9258.

[4] C.-C. Wu, N. B. Dale, and L. J. Bethel, "Conceptual models and cognitive learning styles in teaching recursion," in *Technical Symp. on Comp. Sci. Education*. Atlanta, Georgia, USA: ACM SIGCSE, Mar. 1998, pp. 292–296, ISBN 0-89791-994-7.

[5] D. Levy and T. Lapidot, "Recursively speaking: analyzing students' discourse of recursive phenomena," in *Technical Symp. on Comp. Sci. Education*. Austin, Texas, USA: ACM SIGCSE, Mar. 2000, pp. 315–319, ISBN 1-58113-213-1.

[6] D. Levy, T. Lapidot, and T. Paz, "'It's just like the whole picture, but smaller': Expressions of gradualism, self-similarity, and other pre-conceptions while classifying recursive phenomena," in *Workshop of the Psychology of Programming Interest Group*, Bournemouth, UK, Apr. 2001, pp. 249–262.

[7] D. Levy and T. Lapidot, "Shared terminology, private syntax: the case of recursive descriptions," in *Conf. on Innovation and Technology in Comp. Sci. Education*. Aarhus, Denmark: ACM SIGCSE, Jun. 2002, pp. 89–93, ISBN 1-58113-499-1.

[8] D. Levy, "Insights and conflicts in discussing recursion: A case study," *Computer Science Education*, vol. 11, no. 4, pp. 305–322, Dec. 2001.

[9] D. Ginat, "The suitable way is backwards, but they work forward," *Journal of Computers in Mathematics and Science Teaching*, vol. 24, no. 1, pp. 73–88, Jan. 2005, ISSN 0731-9258.

[10] J. Segal, "Empirical studies of functional programming learners evaluating recursive functions," *Instructional Science*, vol. 22, no. 5, pp. 385–411, Sep. 1994, ISSN 0020-4277.

[11] C. D. Clack and C. Myers, "The dys-functional student," in *Intl. Symp. on Functional Programming Lang. in Education*, ser. LNCS, no. 1022. Springer-Verlag, London, UK, 1995, pp. 289–309, ISBN 3-540-60675-0.

[12] F. Turbak, C. Royden, J. Stephan, and J. Herbst, "Teaching recursion before loops in CS1," *Journal of Computing in Small Colleges*, vol. 14, no. 4, pp. 86–101, May 1999.

[13] M. Rubio-Sánchez and I. Hernán-Losada, "Exploring recursion with Fibonacci numbers," in *Conf. on Innovation and Technology in Comp. Sci. Education*. Dundee, Scotland, UK: ACM SIGCSE-SIGCUE, Sep. 2007, pp. 359–359, ISSN 0097-8418.

[14] D. Wilcocks and I. Sanders, "Animating recursion as an aid to instruction," *Computers & Education*, vol. 23, no. 3, pp. 221–226, Nov. 1994, ISSN 0360-1315.

[15] T. Rosenthal, "Introducing recursion by using multimedia," in *Conf. on Innovation and Technology in Comp. Sci. Education*. Caparica, Portugal: ACM SIGCSE, Jun. 2005, pp. 374–374, ISBN 1-59593-024-8.

[16] B. Stephenson, "Visual examples of recursion," in *Conf. on Innovation and Technology in Comp. Sci. Education*. Paris, France: ACM SIGCSE, Jul. 2009, pp. 400–400, ISBN 978-1-60558-381-5.

[17] C.-C. Wu, G. C. Lee, and J. M.-C. Lin, "Visualizing programming in recursion and linked lists," in *Australasian Conf. on Comp. Sci. Education*, ser. ACM Intl. Conf. Proc., vol. 3. University of Queensland, Australia: ACM SIGCSE, Jul. 1998, pp. 180–186, ISBN 1-58113-018-X.

[18] J. Kelso, "A visual programming environment for functional languages," Ph.D. dissertation, Murdoch University, Perth, Australia, 2002.

[19] C. D. Hundhausen, S. F. Farley, and J. L. Brown, "Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study," *ACM Trans. on Computer-Human Interaction*, vol. 16, no. 3, Sep. 2009, ISSN 1073-0516.