THE SLIPPERY ROAD FROM ACTIONS ON OBJECTS TO FUNCTIONS AND VARIABLES

Tamar Paz and Uri Leron

Technion – Israel Institute of Technology

Functions are all around us, disguised as actions on concrete objects. Furthermore, composition of functions too is all around us, because these actions can be performed in succession, the output of one serving as the input for the next. This action-on-objects scheme can serve as excellent entry point for learning about functions, as for example in the function machine metaphor, but it can also lead to some deep misconceptions. As our data shows, the intuitions about change and invariance entailed by the action-on-objects scheme may clash with the modern concepts of function and of composition of functions.

1. Introduction

Functions are all around us in everyday life; or at least a mathematically trained person can see functions all around. The actions of moving objects about (including one's body), squeezing a piece of clay, rearranging pebbles, or commanding the Logo turtle to go *forward 100*, can all be seen as functions, because they clearly have an input (initial state) and a well-defined output (final state). Furthermore, composition of functions too is all around us in everyday life, because these actions can be performed in succession, the output of one serving as the input for the next

In this article we investigate the relationship between the intuitions of actions on familiar objects on the one hand, and the formal function concept on the other. We focus on the "algebraic" conception of functions, which is based on the image of an input-output machine, as opposed to the "analytic" conception, which is based on the image of covariation of two magnitudes. In terms of the theoretical framework of Lakoff & Núñez's (2000) *mathematical idea analysis*, we propose that the function concept, at least in its algebraic conception, is metaphorically grounded in these intuitions of actions on objects. This idea will be developed in the first part (2.1) of the theoretical background. In the second part (2.2) we will introduce an influential contemporary theory of intuitive thinking from cognitive psychology, called *dual process theory*, which will help us interpret our data. In the third part of the theoretical background (2.3), we will briefly introduce the *functional programming paradigm*, which helped us uncover some deep-seated pre-conceptions about functions. In the research part (section 3), we bring data to demonstrate our main thesis, namely, that the same intuitions of actions on objects can also stand in the way of further understanding of functions. We will also interpret some of our findings through the perspective of dual process theory.

2. Theoretical Background

2.1. How functions can be conceived (and misconceived) as actions on objects

One of our most basic intuitions is perceiving the (physical) world in terms of objects and actions on them. Indeed, according to Piaget (1983/1970) this is one of the fundamental mechanisms through which the developing child comes to know the world:

> Actually, in order to know objects, the subject must act upon them, and therefore transform them: he must displace, connect, combine, take apart, and reassemble them. From the most elementary sensorimotor actions (such as pulling and pushing) to the most sophisticated

> intellectual operations, which are interiorized actions, carried out mentally (e.g., joining together, putting in order, putting into one-to-one correspondence), knowledge is constantly linked with actions or operations […]. (p. 104)
>
> Indeed, if logic and mathematics are so-called "abstract" sciences, the psychologist must ask, Abstracted from what? […] Therefore the origin of these logicomathematical structures should be sought in the activities of the subject, that is, in the most general forms of coordinations of his actions, and, finally, in his organic structures themselves. This is the reason why there are fundamental relations among the biological theory of adaptation by self-regulation, developmental psychology, and genetic epistemology. This relation is so fundamental that if it is overlooked, no general theory of the development of intelligence can be established. (p. 106)
>
> [Logicomathematical] experience also involves acting upon objects, for there can be no experience without action at its source, whether real or imagined, because its absence would mean there would be no contact with the external world. (p. 118)

Thus, a child learns to make sense of the world by interacting with its environment, specifically, by gradually coming to perceive objects in its environment and by performing various actions on these objects. We shall refer to this universal trait as the *Actions-On-Objects Scheme* (AOS). The essence of the AOS for us – and the one we will invoke in explaining our findings about functions – is the conception that *when an action has been performed on an object, the object has undergone some change, but it is still* ***the same*** *object before and after the operation*. Or, in Piaget's (1983/1970) words: "[All children agree that] when a ball of clay is changed into a sausage, it is the 'same' lump of clay even if quantity is not preserved." (p. 123)

We note that this conception may not apply in some extreme cases, such as when a glass is smashed into splinters. It is enough, however, that some essential property of the object is transformed continuously, as can be seen from the example[1] of the frog, which upon being kissed turns into a prince: The input-frog and the output-prince are perceived as "the same being", despite the drastic transformation.

Actions on familiar objects also influence our intuition about mathematical functions, at least in their "algebraic" conception, that is, when viewed as input-output machines (as opposed to the "analytic" conception, where the intuition comes more from the notion of covariation of two magnitudes). But as the function concept evolved towards more and more formal and abstract definitions, it has also become more and more distanced from these intuitions and, as we will show, the image of actions on concrete objects has sometimes turned from being a support into being an obstacle to the understanding of the modern function concept.

The empirical study of the relationship between functions and the AOS (of which this study is a mere beginning) falls under the branch of cognitive science that Lakoff & Núñez (2000) call *Mathematical idea analysis*: [2]

> For the most part, human beings conceptualize abstract concepts in concrete terms, using ideas and modes of reasoning grounded in the sensory-motor system. The mechanism by which the abstract is comprehended in terms of the concrete is called *conceptual metaphor*. Mathematical thought also makes use of conceptual metaphor, as when we conceptualize numbers as points on a line. (p. 5)

---

[1] Thanks to Lisser Rye Ejersbo.

[2] Though we are unconvinced by many of Lakoff & Núñez's specific analyses of mathematical concepts, we nonetheless find the general framework of mathematical idea analysis useful and important.

Lakoff & Núñez (2000) also discuss briefly some of the conceptual metaphors for functions (e.g. on p. 386), but not for the algebraic conception, which is our concern here. In our case, to be specific, the metaphorical mapping would map *action* to *function*, *object* (or the *state* of the object) to *variable*, and the *initial* and *final state* of the transformed object to the function's *input* and *output*.

Our empirical findings below demonstrate the influence of the AOS – in particular the conception mentioned above that the object is changed but still remains the same – on students' conception of functions. In a way, these findings help substantiate the theoretical analysis of the function concept as grounded in the AOS metaphor. On the one hand, teachers and textbooks commonly use some version of the AOS to engage students' intuitions when thinking about functions; witness, e.g., the prevalent use of the function machine metaphor, or expressions like "the function that multiplies x by 2" (see more examples below, in the context of functional programming). On the other hand, as we will show, this same image clashes with the formal function concept, since it invites the erroneous conclusion that if we apply the above function to (say) x = 3, then x will become 6. The general AOS-induced image is that the input-object is being transformed by the function into the output-object, but still it remains "the same" object.

2.2. Dual process theory

A substantial part of mathematics and CS education research is concerned (explicitly or implicitly) with the relationship between the intuitive and analytical modes of thinking and behavior (e.g., Fischbein, 1987; Stavy and Tirosh, 2000). Empirical findings on misconceptions are often explained by the mismatch between students' intuitions and the formal requirements of the modern mathematics.

Recent research in cognitive psychology demonstrates that people consistently make mistakes on simple everyday tasks, even when the subjects are knowledgeable, intelligent people, who may actually posses the necessary knowledge and skills to perform correctly on those tasks. This research – the *heuristics and biases program* – has been carried out by Kahneman & Tversky and others during the last thirty years, and led to Kahneman's receiving the 2002 Nobel Prize in economics[3].

One popular interpretation of this research is that people behave *irrationally*, because they answer incorrectly. But other researchers maintain that there is no reason why people should behave in everyday situations according to the *norms* of mathematics, logic or statistics. On this view, people's answers are *non-normative* rather than incorrect or irrational. We will not pursue this fascinating *rationally debate* here (see e.g. Stein, 1996), but will adopt the terms *normative* to emphasize that whether the answers are judged correct or incorrect depends on the norms one chooses to adopt.

In his Nobel Prize lecture, Kahneman opened with the following story:

> **Question:** A baseball bat and ball cost together one dollar and 10 cents. The bat costs one dollar more than the ball. How much does the ball cost?
>
> Almost everyone reports an initial tendency to answer '10 cents' because the sum $1.10 separates naturally into $1 and 10 cents, and 10 cents is about the right magnitude. Frederick found that many intelligent people yield to this immediate impulse: 50% (47/93) of Princeton students and 56% (164/293) of students at the University of Michigan gave the wrong answer. (Kahneman 2002, p. 451. See also Kahneman and Frederick, 2005, p. 273.)

What are our mind's mechanisms that may account for these empirical findings? One current influential model in cognitive psychology is *Dual process Theory* (Kahaneman, 2002; Stanovich &

[3] Tversky unfortunately died several years earlier.

West 2000, 2003). According to this theory, our cognition and behaviour operate *in parallel* in two quite different modes, called System 1 (S1) and System 2 (S2), roughly corresponding to our common sense notions of intuitive and analytical thinking. These modes operate in different ways, are activated by different parts of the brain, and have different evolutionary origins (S2 being evolutionarily more recent and, in fact, largely reflecting *cultural* evolution). The distinction between perception and cognition is ancient and well known, but the introduction of S1, which sits halfway between perception and (analytical) cognition, is relatively new, and has important consequences for how empirical findings in cognitive psychology are interpreted, including application to mathematics and computer science (CS) education research.

Like perception, S1 processes are characterized as being fast, automatic, effortless, "cheap" in terms of working memory resources, unconscious and inflexible (hard to change or overcome); unlike perception, S1 processes can be language-mediated and relate to events not in the here-and-now (i.e., events in far-away locations and in the past or future). In contrast, S2 processes are slow, conscious, fully engage the working memory resources, and relatively flexible. In addition, S2 serves as monitor and critic of the fast automatic responses of S1, with the "authority" to override them when necessary. In many situations, S1 and S2 work in concert, but there are situations in which S1 produces quick automatic *non-normative* responses, while S2 may or may not intervene in its role as monitor and critic. See the heuristics-and-biases research by Kahneman & Tversky (e.g., Kahneman, 2002; Kahneman & Frederick, 2005).

A brief analysis of the bat-and-ball data can demonstrate the usefulness of dual-process theory for interpretation of empirical data. According to this theory, we may think of this phenomenon as a "cognitive illusion", analogous to the famous optical illusions from cognitive psychology. The surface features of the problem cause S1 to jump immediately with the answer '10 cents', since the numbers 'one dollar' and '10 cents' are salient, and since the orders of magnitude are about right. The roughly 50% of students who answer '10 cents' simply accept S1's response uncritically. For the rest, S1 also jumps immediately with this answer, but in the next stage, S2 interferes critically and makes the necessary adjustments to give the correct answer ('5 cents').

Recently an analogous phenomenon has been demonstrated with respect to mathematical thinking. Leron and Hazzan (in print) found that college students learning abstract algebra exhibit the same behavior. The interesting element in this discovery is that while it seems natural that in everyday situations people should prefer quick approximate responses that come easily to mind over careful systematic rule-bound calculations, students solving mathematical problems during a university course would be expected to consciously train their methodological thinking to check, and override if necessary, their immediate intuitive responses. From these findings we may understand the strong influence intuition, especially its tendency to be influenced by surface clues, has on our thinking.

2.3. A brief introduction to functional programming

The discipline of computer science comprises several major programming paradigms. The research described below focuses on Israeli high school students learning the functional programming paradigm, where the basic entities are functions and composition of functions. This approach is usually studied as an "additional paradigm", after or in parallel to studying the procedural paradigm in Pascal. The functional paradigm has originated with the LISP programming language and its various offsprings, including later dialects such as Logo and Scheme – the language used by the research population. In these languages the basic data objects are lists, i.e. an ordered set of objects of the language. For example, the following is a list with 4 elements (the last being itself a list): *(all we need (is love))*. We can create *variables* in the language by assigning a name to an object. For example, suppose the name *L* is assigned to the above list. Then a variable has been created whose *name* is *L* and its *value* is the list *(all we need (is love))*.

In addition to lists, functional programming consists of operations (functions) on lists. For example, the operation *first* inputs a list and outputs its first element; thus, for the list *L* defined above, *(first L)* will output the word *all*.[4] Similarly, *rest* inputs a list and outputs the list without its first element; thus *(rest L)* will output the list *(we need (is love))*. Finally, *cons* inputs any object and a list and adds the object to the beginning of the list. More precisely, *cons* constructs a new list whose *first* is the input object and whose *rest* is the input list. Two functions can be *composed*, as in mathematics, by taking the output of one as the input to the other; thus, *(first (rest L))* will output *we*. Similarly, (*cons (first L) (rest L))* will output the original list *L*.

We remark that in order to ensure that students experience a truly different way of thinking and programming, especially emphasizing the difference from Pascal, a didactical decision was made to avoid the use of direct assignment (similar to "let x equal 3" in mathematics, or x := 3 in Pascal). Students still worked a lot with *indirect* assignment through the input variables of functions.

A simple example will illustrate the difference between the procedural style with its emphasis on assignment, and the functional paradigm with its emphasis on functions and function composition.

The task: Given a function, called *max2* which inputs two numbers and returns their maximum, write a function, *max3*, which returns the maximum of *three* numbers.

Here are typical solutions in the two paradigms, both implementing the same algorithm, namely, computing *max3* by applying *max2* twice in succession:

Procedural paradigm (Pascal)	Functional paradigm (Scheme)
Function max3 (x , y , z : real) : real; *Var m2 : integer;* *begin* *m2 := max2 (x , y);* *max3 := max2 (m2 , z);* *end;*	*(define (max3 x y z)* *(max2 (max2 x y) z))*

This example shows one of the difficulties students of the functional paradigm experience because of the decision (justified though it is) not to allow the use of direct assignment. The function on the left is less elegant mathematically but easier for students because of the ability to name and store intermediate results. While the functional expression *(max2 (max2 x y) z)* is mathematically more elegant, our research indicates that it is easier for students to store in their memory the intermediate result as *m2* rather than as *(max2 x y)*.

The above example also shows how variables in Scheme can still be created and assigned values even though students do not have access for a direct assignment command. For example, in order to calculate the maximum of 5, 2 and 9 using the above function *max3* we write the instruction *(max3 5 2 9)*, whereby the values 5, 2 and 9 get assigned, respectively, to the variables *x*, *y* and z.

[4] Note the slight difference in syntax: In Scheme we write *(first L)* instead of the customary *first(L)* from mathematics.

3. Research finding: The clash between the AOS and the function concept

The study population consisted of five 11-grade classes (about 20 students each), who studied functional programming in a DrScheme environment.[5] Observations were also made in additional classes selected from three schools in Northern Israel.

We use a qualitative research approach, characterized by a flexibly developing research setup, gradual refinement of the research foci, and parallel processes of data collection and analysis. The research data, collected mainly through observations and interviews, were analysed in order to draw conclusions and develop a "grounded theory" that would reciprocally inform the subsequent course of research. During the initial stages of the research, classroom observations were the main tool for data collection. Later on, special interviews were conducted with students, in which the students were observed and taped as they were working on programming tasks or asked to explain given functions.

From a general perspective, this paper can be viewed as contributing additional evidence to the familiar and prevalent phenomenon of a clash between people's intuitions and the cognitive requirements of contemporary formal systems, such as mathematics, science, and computer science (e.g., Fischbein, 1987; Stavy & Tirosh, 2000; Geary, 2002; Leron & Hazzan, in print). Specifically, we study this clash in the case of the function concept in mathematics and computer science. As mentioned above, we propose that in certain circumstances the intuitions coming from the Actions-On-Object Scheme (AOS) clash with the formal function concept. Our data demonstrates two kinds of such clashes: One (discussed in 3.1 below), the clash between the AOS intuition of *change* vs. the formalism of function as *correspondence*: Under the influence of the AOS, the students perceive the function as changing the input-object into the output-object, yet in the formal function concept nothing really changes – the output-object simply corresponds to the input-object, but itself remains unchanged. Two (3.2), the clash between the AOS intuition of performing a *chain of actions* on the same (continuously changing) object, vs. the *composition* of functions.

3.1. Do functions change their input?

Following are two cases of a typical phenomenon we have frequently observed among our students.

Case 1: In the following task, Dan was asked to explain the behavior of the function *last-in-second*, which inserts the last element of the input list in the list's second place (The explanation on the right-hand column is for readers' benefit, and was not given to Dan).

The programming code	Explanation
(define (last-in-second L)	Define a function called *last-in-second* with a list *L* as input variable.
(cons (first L) (cons (last L) (rest L))))	Build a list from (in this order): the first element of *L*, the last element of *L*, and *L* without its first element.

Dan: So in the function that I define, the list changes each time. My *L* is not the original *L* […] When I do *rest L* then the *L* is not (1 2 3 4 5 6), it is something like this [erases the 1 and points to (2 3 4 5 6)].

[5] DrScheme has been developed at Rice University, USA, with the objective of offering a pedagogical environment for functional programming (Felleisen et al., 2001). The software can be downloaded from http://download.plt-scheme.org/drscheme/ .

Dan is correctly trying to trace the evaluation of the expression from right to left. He is looking at the situation after *rest L* and *last L* (which poses no problem) have been executed and is trying to figure out the value of *first L*, specifically, the value of *L* that *first* gets as input. Clearly, he believes that *rest* has actually chopped off the first element of *L*.

Case 2: Mili was working on the following task:

> *Write a function which inputs a list* L *and a number* x, *and inserts the number both before and after the first element of* L. *For example, if the input list is* (a b c) *and the number is* 7, *the output should be the list* (7 a 7 b c) .[6]
>
> Mili wrote in her notebook *rest L* and stopped. Then she called the researcher for help.
> Researcher: What's the problem?
> Mili: I ask is it possible to do *first L* and then I get, like, *a*?
> Researcher: Is *first L* something the computer knows how to do?
> Mili: Yes.
> Researcher: So what's the question?
> Mili: Is it allowed?
> Researcher: Why shouldn't it be allowed to do *first L*?
> Mili: But no, because I already chopped off the list. The question is, is it allowed to do it again to the full list?
> Researcher: When you did *rest* here, does it mean that the list was spoiled?
> Mili: Yes.
> Researcher: Why?
> Mili: Because it is now left with only *b* and *c*.

From these two examples, it is clear that the students think that *a function changes its input,* indeed that the function's output becomes the new value of its input variable. Thus, Dan thinks that *rest* chops off the first element of a list; namely, if *L* = (1 2 3 4 5 6), then after *rest L* has been executed, *L* becomes (2 3 4 5 6). Similarly, Mili thinks that if *L* is *(a b c)*, then after *rest L* has been executed, *L* becomes *(b c)*. Needless to say, this is not what really happens. In modern mathematics and computer science the input value *corresponds* to the output value, but nothing really changes; and in the case of programming, this conception leads to programming errors. We emphasize that this intuition is very strong and very prevalent: it was found in many students and in every class, and even among mature students and computer science teachers. In our interpretation, this is a case of the clash between the AOS – which leads to the view of function as an agent of change – and the formal definition of function which denies this view.

In terms of the dual process theory, we could say that AOS is part of S1, and is here running the show, and S2 lacks the knowledge to override this result in favor of the normative answer. That is, in this case S1 and S2 together adopt the AOS scheme, and give a non-normative response. We will have an opportunity to use of the dual process theory in a more substantial way in the next section, where S1 and S2 hold conflicting views.

The students in our study may have been influenced by additional factor. Many of the operations on variables in the Pascal programming language, with which our students were familiar, change the variable on which they operate. Examples are *read(num)* which stores the value of its input in the variable *num*, and *count := count + 1*, which increments the variable *count* by 1. However, we will not pursue this explanation further in this article.

---

[6] Typical solution: *(cons x (cons first L (cons x rest L)))* .

Some readers may feel that the formulation of the task (the function *inserts the number* in the list *L*) may have induced the misconception, since it actually describes the function as changing its input *L*. Indeed, a more precise (but more clumsy) formulation would be "a function that input a list and a number, and outputs a new list which is identical to the input list except…". Yet, most experienced teachers prefer the first formulation, precisely because they prefer intuitive formulations. This is also true of textbooks (e.g., Harvey & Wright, 1994; Felleisen et al., 2000), including the textbook that our subjects learned form. Incidentally, the same formulation is also common in mathematics teaching, as in the example of a function machine that takes in a number and "multiplies it by 2".

More generally, math and science teachers like to use metaphors from everyday life in order to engage the student's intuition and motivation, even though such metaphors always have a limited "scope of validity", and are likely to clash with the rigorous scientific concepts outside this scope (e.g., "the computer doesn't understand", or "the selfish gene"). The rational for such use is that the metaphor is very useful in the beginning (Lakoff and Johnson, 1980), before the students are ready to deal with the subsequent subtleties. In addition, much research from mathematics education and from cognitive psychology indicates that such intuitive 'biases' are very robust in the face of explanations and change of formulation. We do not believe that choosing the more precise formulation would help avoid this 'bias'. Instead, we believe that the changing-the-input bias itself should be discussed with the students when the problem actually comes up, drawing their attention to the clash.

3.2. Chaining or composition?

In teaching programming of complex tasks, we use the technique of first asking students to describe the algorithm in natural language. This encourages them to use their intuition and everyday experience, and usually they have no problem coming up with the desired description. But now the question arises, how do we go from here to the formal code? One step is translating from natural language to programming code, and often this is all that's necessary, especially in procedural languages (such as Pascal), where direct assignment is routinely used. In our functional programming classes, however, direct assignment was not available to the students (as explained above), and the translation has not been so straightforward. Specifically, when the verbal description consists of a chain of actions on objects, this cannot be translated intuitively as a chain of assignments, but has to be formulated as a composition of functions, which is known to be hard. We examine two cases of this phenomenon.

Case 1: Sharon was working on the following task:

Write a function which inputs a number and a list, and replaces the last element of the list with the number. For example, if the number is 6 *and the list is* (a b c)*, then function should output the list* (a b 6)*.*

In describing the algorithm in natural language, the first thing the students would want to do is remove the last element of the list. However, the given programming language has an instruction (*rest*) for removing the first element of the list, but not one for the last. What one usually does instead is reverse the list and then remove the first element. The remaining actions in the present case are to add the new number to the beginning of the reversed list and reverse again. Note that this is a description within the AOS since it is given in terms of actions on concrete objects.[7] The translation to functional programming code (which is the same as mathematical formalism except for slight syntactical differences) is at the heart of this paper, since it shows how students negotiate the gap between intuitive actions and formal functions. Here is how Sharon managed this translation.

---

[7] Concrete is a relative term, and a list of numbers or words is very concrete for these students.

Sharon's code	Explanation
(define (replace-last n L)	Define a function called *replace-last*, with a number *n* and a list *L* as input variables.
(reverse L)	Reverse the list
(rest L)	Remove the first element
(cons n L)	Add n in the beginning of the list
(reverse L))	Reverse the list

This indeed is a precise rendering in the programming language of the above AOS description, but unfortunately it doesn't work. Sharon, presumably guided by the intuition of the AOS, writes a chain of actions, believing that each works on the result of the previous one (as in real life). However in functional programming, as in mathematics, what is needed here is composition of functions, not chaining, thus:

(reverse (cons n (rest (reverse L)))) .

This demonstrates another facet of the clash between AOS and the modern concept of function; that is, the intuition of chaining actions on objects clashes with the required formalism of composition of functions. We will have more to say about Sharon's data after the next example.

Case 2: Here we consider an even stronger evidence for the same phenomenon, though in the context of a more complicated task. The task consisted of two parts, the first intended to serve as preparation for the second.

(a) *Write a function which inputs a list and returns the first element which is a number.* For example, if the input list is (Helen is 17 years old and Ben is 15 years old) then the function returns 17.

(b) *Write a function which inputs a list and returns the last element which is a number.* For example, if the input list is (Helen is 17 years old and Ben is 15 years old) then the function returns 15.

We used this task in various forms on many occasions, some with students and some with teachers. Part (a) has a standard solution using recursion, and was easily solved by most students and teachers alike. Part (b) is more tricky because of the limitation of the language that it has a *first* command but not *last*. The solution of (a) is straightforward:

The programming code	Explanation
(define (first-number L)	Define a function called *first-number* with a list *L* as input variable.
(cond	Checking the condition:
((number? (first L)) (first L))	If the first element is a number, return this element
(else (first-number (rest L)))))	If not, apply the same function recursively on the list with its first element removed

Part (b), in contrast, is more tricky, and is not easy for our students to solve. There are basically two ways to approach (b), each of them with its own difficulties, as we now explain. The first (and most elegant) solution is as follows.

The programming code	Explanation
(define (last-number L)	Define a function called *last-number* with a list *L* as input variable.
(first-number (reverse L)))	Applying the previous *first-number* function to the reversed list

This seemingly-simple solution, which comes easily and naturally to an expert, is not at all easy for a beginner, and is not usually found by our students or student-teachers during their first course. In our interpretation, the idea of using the function *first-number* from part (a) doesn't occur to the students – despite having programmed it themselves! – because they haven't encapsulated the function into a single entity. A supporting evidence for this explanation is the fact that the same students can use composition of functions in other situations with no difficulty as can be seen for example in their solution of part (a). This phenomenon is similar to the one described by Schoenfeld (1985, pp. 181-182), where students fail to use a lemma which they themselves proved in part (a) of a geometry task, which would have rendered part (b) almost trivial.

Instead of *reducing* the solution of (b) to that of (a) as we just did, one can alternatively *imitate* the solution of (a) by performing the same steps on the reversed list. This solution, which is perceived by students as more straightforward, leads in turn to non-trivial technical and conceptual difficulties. It is not easy, for example, to understand why *reverse* should appear in the last line *twice*. Here is how such a solution might look:

```
(define (last-number L)
  (cond
    ((number? (first (reverse L)))  (first (reverse L)))
   (else  (last-number  (reverse (rest (reverse L)))))))
```

In the case we are about to discuss, the task was given as homework to a group of computer science teachers, participating in an introductory functional programming course. The participants were 21 experienced high school teachers, who had been teaching for the matriculation exams in computer science (with the Pascal language) for at least 5 years. One of the teachers, Emma, who had solved part (a) correctly, presented to the class the following solution of part (b), saying: "I have a solution, I checked it many times and I don't understand why it doesn't work":

```
(define (last-number L)
   (reverse L)
  (cond
    ((number?  (first L))  (first L))
    (else  (last-number  (rest L)))))
```

Emma also supplied the following explanation for her solution:

> First it does *reverse* and then on the reversed list we need to do what we did in *first-number*, so I copied the instructions of *first-number*.

Emma's initial idea was correct: Reverse the list and then imitate the solution of part (a). This is the same idea behind our own second solutions above. The problem arises, as in Sharon's case, in translating this idea from natural language to the computer language. As in Sharon's case, we observe here the two AOS-induced problems: chaining operations instead of composing functions, and the functions-change-their-input behavior (in this case, the assumption that after executing *reverse L*, *L* has become the reversed list). In practice, the computer will first evaluate *reverse L*

doing nothing with the result, then carry out the conditional expression on the original list *L*, in effect computing *first-number L*.[8]

In the present case there was an additional complication (and a corresponding additional programming error) stemming from the use of recursion. The problem is that each recursive call executes all the instructions in the definition of the function, hence *reverse L* would be executed again in every recursive call, not just at the first call as Emma intended. This error reflects a problem in understanding recursion, not functions, but, as will be seen, there is interference between the two kinds of understandings due to general cognitive mechanisms.

In two minds

The last two examples highlight an interesting phenomenon. Sharon seems to assume that, like real-world objects, *L* is constantly changed, each *L* being the result of the previous operation. Similarly, the participating teachers in the present scenario were completely oblivious to the changing-the-parameter and the chaining problems in Emma's solution, being instead immersed in a prolonged class discussion on what the recursive call would do to the reversed list. From our acquaintance of these teachers and their performance on other tasks, we know that they are in full possession of all the knowledge necessary to avoid these function-related errors. Indeed, when given more elementary tasks, they have written correct programs dealing with recursion and with composition of functions, and they have demonstrated the knowledge that functions do not change their inputs. This shows that the students' (and teachers') learning during the course did not eliminate the intuitions of the AOS, just drove them underground. When they were working on a complicated task that demanded their full attention, the AOS intuitions re-surfaced and took control of their performance.

This phenomenon can be neatly explained in terms the *dual process theory* as introduced in section 2.2. In terms of dual process theory, we might say that the intuitions of the AOS had initially been part of the students' S1 knowledge, and remained so even after the students learned that functions do not change their inputs – the latter having become part of their S2 knowledge. (It is quite common for S1 and S2 to simultaneously hold conflicting pieces of knowledge.) Since S2 relies heavily on the limited resources of working memory, it is less likely to do its S1-monitoring job while being engaged in another complex task, as explained in Kahneman and Frederick's (2005):

> The effect of concurrent cognitive tasks provides the most useful indication of whether a given mental process belongs to system 1 or system 2. Because the overall capacity for mental effort is limited, effortful processes tend to disrupt each other, whereas effortless processes neither cause nor suffer much interference when combined with other tasks […] People who are occupied by a demanding mental activity […] are much more likely to respond to another task by blurting out whatever comes to mind […]. (p. 268)

We can now offer the following dual process interpretation of the present case: Usually, when given a task that directly tests the students' knowledge of whether a function changes its input, the fast automatic S1 response (that it does) first comes to their mind, but is quickly overridden by their S2 knowledge (that it doesn't). This is similar to the analysis of the bat-and-ball task in section 2.2. In the case reported here, the students' S2 was busy working on the complex recursive task, hence S1 could "hijack" their performance with its automatic intuitive response, without being "caught" and corrected by the busy S2.

---

[8] The precise details may differ between different functional dialects, but these differences do not affect the present analysis.

4. Conclusion

We have seen that the AOS can serve as an effective intuitive support for learning about functions, but that it can also clash with the formal function concept, leading to some persistent programming errors. We believe that more research may reveal similar phenomena in other mathematical concepts, such as limits and continuity. We conjecture that this is typical of the (cultural) evolution of mathematical concepts: While they may have their origin in everyday intuitions (Lakoff & Núñez, 2000), their modern formal incarnation may often clash with these very same roots. We further propose that the reason for this clash is that the modern version, in order to achieve utmost rigor, power, and consistency, has to suppress all traces of time and process. But our basic intuitions about the world are rooted in action, hence are inseparable from the very same processes that are eliminated in the modern formalism.

5. References

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamuethi, S. (2001). *How to Design Programs: An Introduction to Computing and Programming*. The MIT Press.

Fischbein, E. (1987). *Intuition in Science and Mathematics: An Educational Approach,* Reidel.

Geary, D. (2002). 'Principles of evolutionary educational psychology'. *Learning and Individual Differences 12*.

Harvey, B., & Wright, W. (1994). *Simply Scheme. Introducing Computer Science*. Cambridge, Massachusetts: The MIT Press.

Kahneman, D. (2002). 'Maps of bounded rationality: A perspective on intuitive judgment and choice', Nobel Prize Lecture. In Frangsmyr, T. (Ed.), *Les Prix Nobel*, 416-499. Web: http://www.nobel.se/economics/laureates/2002/kahnemann-lecture.pdf.

Kahneman, D., & Frederick, S. (2005). 'A Model of Heuristic Judgment'. In Holyoak, K.J. & Morrison, R.J. (Eds.), *The Cambridge Handbook of Thinking and Reasoning*, pp. 267-293. Cambridge University Press.

Lakoff, G. and Johnson, M. (1980), *Metaphors we live by*. The University of Chicago Press.

Lakoff, G., & Núñez, R. (2000). *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics Into Being*, Basic Books.

Leron, U., & Hazzan, O. (in print). 'The Rationality Debate: Application of Cognitive Psychology to Mathematics Education'. *Educational Studies in Mathematics*.

Piaget, J. (1983/1970). 'Piaget's Theory'. In P. H. Mussen (Ed.), *Handbook of Child Psychology, Fourth edition*, Vol. 1, pp. 103-128. Wiley.

Schoenfeld, A.H. (1985). *Mathematical Problem Solving*. Orlando, Florida: Academic Press, inc.

Stanovich, K. E., & West, R. F. (2000). 'Individual differences in reasoning: Implications for the rationality Debate'. *Behavioral and Brain Sciences*, 23, 645–726.

Stanovich, K. E., & West, R. F. (2003). 'Evolutionary versus instrumental goals: How evolutionary psychology misconceives human rationality'. In Over, D. E. (Ed.). *Evolution and the Psychology of Thinking: The Debate*, Psychology Press, pp. 171-230.

Stavy, R., & Tirosh, D. (2000). *How Students (Mis-)Understand Science and Mathematics: Intuitive Rule*. Teachers College Press.

Stein, E. (1996). *Without Good reason: The Rationality Debate in Philosophy and Cognitive Science*, Oxford