

Running Head: **THE SLIPPERY ROAD**

**The Slippery Road from Actions on Objects
to Functions and Variables**

Tamar Paz and Uri Leron

Technion – Israel Institute of Technology

Abstract

Functions are all around us, disguised as actions on concrete objects. 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. In terms of Gray and Tall's 'embodied objects' or Lakoff and Núñez's 'mathematical idea analysis', this 'embodied scheme' of action on objects may serve as intuitive grounding for the function concept. However, as Gray and Tall and their colleagues have shown, such embodied schemes can also lead to serious 'epistemological obstacles' in later stages of concept development. In the same vein, our own data shows that the intuitions about change and invariance entailed by the action-on-objects scheme, while helpful in earlier stages of learning functions, may later come to clash with the formal 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 a child moving objects about – squeezing a piece of clay, rearranging pebbles, commanding the Logo turtle to go *forward 100*, or moving her own body – 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, 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. This naturally leads us to focus our discussion on the conception of functions as input-output machines, which applies to all kind of objects, both from everyday life (e.g., moving one's body or rearranging an array of pebbles), from computer science (e.g. reversing the elements of a list), or from advanced mathematics (e.g., symmetries, permutations or homomorphisms). This focus excludes other conceptions of functions such as the covariation of two magnitudes, especially real-valued functions, such as polynomial, trigonometric or exponential functions. In terms of the theoretical frameworks of Gray and Tall's (2001) 'embodied objects' or of Lakoff and Núñez's (2000) 'mathematical idea analysis',¹ we propose that the function concept, at least in its conception as input-output machine, may be metaphorically grounded in the 'embodied scheme' of actions on objects.²

This idea is developed in Section 2.2 of the theoretical background. In Section 2.3 we introduce an influential contemporary theory of intuitive thinking from cognitive psychology, called *dual process theory*, which will help us interpret our data. This is followed (Section

2.3) by a brief introduction to the *functional programming* paradigm, where some deep-seated pre-conceptions about functions were uncovered. In the remaining sections of the theoretical background we review previous research on learning functions (2.1), and the similarities and differences between functions in mathematics vs. computer science (2.5). In the research part (Section 3), we bring data to demonstrate our main thesis, namely, that the same intuitions of actions on objects that are helpful in the initial grounding of the function concept, can later stand in the way of further understanding of the same concept. Finally, in the Conclusion (Section 4), we consider some generalizations of these findings, and some possible classroom applications.

2. Theoretical Background

2.1. Previous research on learning functions

There is a wide agreement that the function concept is difficult to learn, as the following quote typically demonstrates: “[The function concept] proves to be one of the most difficult concepts to master in the learning sequence of school mathematics.” (Eisenberg, 1991)

However, the findings reported here, as well as our theoretical framework, differ from previous research on functions, both because of our emphasis on the input-output conception of functions, and because of the unusual setting (functional programming) that was needed to uncover those findings.

Some of the main difficulties in learning the function concept reported in previous research concerned moving between various representations (Leinhardt, Zaslavsky & Stein, 1990; Eisenberg, 1991; Ponte, 1992; Sierpinska, 1992; DeMarois & Tall, 1999, Carlson &

Oehrtman, 2005), process-object transition (e.g., the various articles in Harel & Dubinsky, 1992), using concept image rather than concept definition (Vinner & Dreyfus, 1989), and confusing the conditions for a function to be single-valued vs. one-to-one (Vinner, 1983; Leinhardt et al., 1990). For extensive reviews of difficulties in learning functions, see Leinhardt, Zaslavsky & Stein, 1990, and Carlson & Oehrtman, 2005.

The focus of this paper is the influence of *everyday experience* on the understanding of functions, in agreement with Leinhardt et al. (1990):

“We define intuitions as features of students' knowledge that arise largely from everyday experience, although in the more advanced student they may involve a mixture of everyday and deeply understood formal knowledge.” (p. 24)

Still, there is not much misconception research on functions which arise from everyday experience. In addition, the school setting naturally directs the research on functions to their conception as covariation of two numerical variables (e.g., Vinner & Dreyfus, 1989), and we haven't been able to find much reference to functions as input-output machines. The work of a few researchers – Piaget, Dubinsky, Lakoff and Nunez, Tall and Gray – which is more closely related to ours, will be discussed in more detail in the next section, in the course of laying out our theoretical framework.

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

One of our most basic intuitions is perceiving the 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 logicmathematical 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 his or her 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 conception – the physical environment as consisting of objects together with the operations they afford – 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 shattered into splinters. It is enough, however, that some essential property of the object is transformed continuously, as can be seen from the example³ 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.

Our use of the AOS should not be confused with Dubinsky's APOS theoretical framework (Dubinsky & McDonald, 2002). The APOS theory (Action, Process, Object, Scheme) is a developmental theory of how mathematical concepts develop during the learning process. In contrast, our AOS framework describes a universal state of primary knowledge even before any teaching has taken place. In particular, we note that our use of the term 'action' is different from Dubinsky's. While we use 'action' in its everyday meaning of what a person is doing when acting on objects (as in the above quotation from Piaget), Dubinsky gives the term a more technical meaning.⁴ The word 'scheme' too is used here in its everyday meaning which in this case is rather similar to Dubinsky's technical definition:

[a] schema for a certain mathematical concept is an individual's collection of actions, processes, objects, and other schemas which are linked by some general principles to form a framework in the individual's mind that may be brought to bear upon a problem situation involving that concept. (Dubinsky & McDonald, 2002, p. 277)

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 and Núñez (2000) call *Mathematical idea analysis*.⁵

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)

Lakoff and Núñez (2000) also discuss briefly some of the conceptual metaphors for functions (e.g. on p. 386), but not as input-output machines, which is our concern here. In our case, to be specific, the metaphorical mapping would map *action* to *function*, *object* (more precisely, the *state* of the object) to the function's *variable*, and the *initial* and *final state* of the transformed object to the function's *input* and *output*. We remark that a metaphorical mapping is inherently vague, since it involves relations between mind entities (a psychological reality) rather than between mathematical objects (a mathematical reality). The mapping is even more vague in our case, due to the ambiguity of the notion of variable even within mathematical discourse. Thus we could just as well map *object* to a *generic element* in the function's domain of definition. These subtleties are interesting in their own right, and deserve further empirical study, but are too fine-grained to affect the analysis in this paper. Our study also fits well within the theoretical framework developed by Gray and Tall (2001) and their colleagues, on the *relationship between embodied objects* (or embodied *schema* in our case) and *symbolic procepts*.⁶ Like the present paper, they point out that these embodied

objects are helpful in early stages of learning, but can cause epistemological obstacles in later stages:

We observe that a practical, real world understanding of simple mathematics can very well benefit from a focus on the operations on the base objects, and such a perspective is satisfactory, even insightful in everyday situations. However, an exclusive focus at this level can act as an epistemological obstacle barring the way to the more sophisticated theory that is required for subtle technical and conceptual thinking. (p. 66)

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” or “the function that rearranges an array of numbers” (as an intuitive introduction to permutations; 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. Thus, if we apply the above function “that multiplies x by 2” to (say) $x = 3$, then this formulation may invite the erroneous conclusion that x would 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. As we will show, this image, if adhered to too literally, may lead to errors in using and in understanding functions.

2.3. Dual process theory

A substantial part of mathematics and computer science 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 & Tirosh, 2000). Empirical findings on misconceptions are often explained by the mismatch between students' intuitions and the formal requirements of modern mathematics.

An important strand in contemporary cognitive psychology also deals with the relationship between the intuitive and analytical modes of thinking and behavior. Research in this strand demonstrates that people consistently make mistakes on simple everyday tasks, even when the subjects are knowledgeable, intelligent people, who may actually possess the necessary knowledge and skills to perform correctly on those tasks. This research – the *heuristics and biases program* – has been carried out by Kahneman and Tversky and others during the last thirty years, and led to Kahneman's receiving the 2002 Nobel Prize in economics.⁷ One popular interpretation of these findings has been that people behave *irrationally*, because they perform incorrectly on simple everyday tasks. But other researchers maintain (on evolutionary or ecological grounds) that we should not expect people to 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 *rationality debate* here (see e.g. Stein, 1996; Samuels et al, 1999; Gigerenzer, 2005), but will adopt the terms *normative* to emphasize that whether the answers are judged correct or incorrect depends on a particular choice of norms.

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

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 & 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* (Kahneman, 2002; Stanovich & West 2000, 2003, Evans, 2008). According to this theory, our cognition and behaviour operate simultaneously 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 midway between perception and (analytical) cognition, is relatively new, and has important consequences for how empirical findings in cognitive psychology are interpreted, including the rationality debate and application to mathematics and computer science 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 and 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. This is amply demonstrated by the heuristics-and-biases research by Kahneman and Tversky and others (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, the irrepressible S1 presumably 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 reported with respect to advanced mathematical thinking. Leron and Hazzan (2006) found that college students learning abstract algebra exhibit similar 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.

2.4. 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 usually appears in the Israeli curriculum as “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 our research population. In these languages the basic data objects are *lists*, i.e. finite ordered sets 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)*). When an instruction or a program is executed, the value of variables can of course be changed during the execution. 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*.⁸ 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 in the initial stages of learning 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 the function *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)	Standard math notation
Function <i>max3</i> (<i>x</i> , <i>y</i> , <i>z</i> : real) : real;	(define (<i>max3</i> <i>x</i> <i>y</i> <i>z</i>)	<i>max3</i> =
var <i>m2</i> : integer;	(<i>max2</i> (<i>max2</i> <i>x</i> <i>y</i>) <i>z</i>))	<i>max2</i> (<i>max2</i> (<i>x</i> , <i>y</i>), <i>z</i>)
begin		

```
m2 := max2 (x , y);  
max3 := max2 (m2 , z);  
end;
```

There is more to this example than just syntax. The example shows one of the difficulties students of the functional paradigm experience because of the decision (justified though it may be didactically) not to allow in the early stages of learning 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 (in the variable $m2$). While the functional expression ($\text{max2} (\text{max2 } x \ y) \ z$) is mathematically more elegant, our experience indicates that it is easier for students to hold in their memory the intermediate result as $m2$ rather than as $(\text{max2 } x \ y)$.

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

2.5 Functions in mathematics vs. functions in computer science

Our paper is about mathematical issues that came up serendipitously during a functional programming course, and not about the educational use of programming to help students learn mathematical functions, although this has been an influential approach in other projects (e.g. Leron & Zazkis, 1989; Dubinsky & Harel, 1992). Thus the following brief comparison

of functions in the two disciplines is intended more as clarification for readers rather than to address the educational issues entailed by such differences. We will still touch briefly on some of these issues.

Before discussing the definitions and use of functions in mathematics and computer science (CS), we note that the very place of definitions is significantly different in the two disciplines. In mathematics formal definitions are required for the rigorous formulation of theorems and proofs. In CS (except for theoretical CS, which can be considered a branch of applied mathematics and is not discussed here), the role of definitions is to establish efficient communication within a community, and no more rigor is exercised than is required for this purpose. Thus, we know precisely what functions mean in math but their meaning in CS is more diffuse, and in addition varies somewhat between different subcommunities.

In this paper we are mainly concerned with the paradigm of functional programming, where a function is usually understood as a procedure with output. The output is usually required to be an object of the programming language under discussion, so that two functions can be composed, the output from one serving as the input to the other. (Thus a procedure that merely prints out its result is not usually considered a function, though there is no problem in thinking of it as a mathematical function. Cf. Leron & Zazkis, 1989, p. 192).

The fundamental view of functions which is shared by math and CS is that of an input/output machine: All functions take in some input object(s) and return an output object.⁹ With this fundamental similarity about the nature of functions in mind, we now note two differences.

One difference follows from the view of functions in CS as procedures with output. In particular, a function is identified with a procedure for computing that function. This

contrasts with mathematics, where a function is specified by its input/output behavior only, ignoring the process (if any) for getting from the input to the output. For example, the functions (geometrical transformations) which rotate the plane by 0 degrees or by 360 degrees, or the composition of two 180-degree rotations, are all considered the same function in mathematics, but would be represented by different procedures (functions) in programming. In situations where this difference matters (e.g., when using programming to teach mathematics), it is easy to deal with the discrepancy by adopting the mathematical criterion for equality of functions, while referring to procedures that compute the same mathematical function as *equivalent* (Leron & Zazkis, 1989, p. 188).

A second (minor) difference is that in CS functions can have different outputs for the same input. Thus we may have a CS function like Random, so that Random(10) will output a different integer between 0 and 9 each time it is executed.

Finally, there are of course notational (syntactic) differences, which are trivial in comparison to the cognitive difficulties involved in coming to understand functions. Thus we do not view them as serious obstacles for learning about mathematical functions.

In general, since definitions in CS are not canonized as in mathematics, it is easy when working in an educational setting to choose (and teach) the precise meaning of function to be what we like, so the vagueness and discrepancies we meet in programming can be turned from a source of confusion into an advantage, namely, a rich source for discussion of subtleties where they really matter. For example, the requirement that the output be specified uniquely for every input becomes more meaningful when the students had the chance in CS to work with some non-examples (Leron & Zazkis, 1989, p. 192).

More fundamentally, the use of programming for learning mathematics, especially in the Logo and ISETL communities (Papert, 1980; Abelson & diSessa, 1981; Dubinsky, 1995) has been based on a ‘constructionist’ perspective: the belief that constructing and manipulating certain objects and relations on the computer will help students construct corresponding objects and relations in their mind. We know that this is at best an effortful and lengthy mental process, even in cases it has been successful. In comparison, dealing with the discrepancies discussed above, seems a minor educational matter.

There is one issue where the difference can matter. Our main claim in this paper is that the AOS influences students' conception of functions. It is possible that this effect is more pronounced in programming than in a purely mathematical context, since the procedural nature of functions in CS may more strongly invoke the AOS. This possibility is worth a further study, and in fact we plan to pursue it in our future research.

3. The Research

3.1 Research setting

The study population consisted of five 11-grade classes (about 20 students each), who studied functional programming in a DrScheme environment.¹⁰ Observations were also made in additional classes selected from three schools in Northern Israel.

The mathematical background of the students was diverse, and involved students with all levels of mathematical ability. What they learned about functions in their middle school and 10th grade math classes had little bearing on the topics discussed here, and consisted solely of functions with numerical input and output, mainly linear and quadratic polynomials. They did

not at all encounter functions of more than one variable, functions with non-numerical variables and composition of functions; neither have they met the function machine metaphor. Thus, it seems reasonable to assume that the intuitions they brought to the functional programming course, did not come from their previous mathematical experience.

The unit on functional programming under discussion consisted of 3 weekly hours for the whole school year, for a total of 90 hours. In some of the classes the course was taken entirely in the computer lab, and in others – 2 weekly hours in the lab and 1 weekly hour in the regular classroom. This unit was the students' second programming course, the first one ("Fundamentals 1") being in the procedural paradigm (Pascal).¹¹ Their prior experience in Pascal included variables, conditional statements, loops, one-dimensional arrays, but not functions. In parallel to the functional programming unit they learned the second procedural module ("Fundamentals 2").

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 theoretical analysis (Goetz & LeCompte, 1984; Denzin & Lincoln, 1994; Bogdan & Biklen, 1998). The research data, collected mainly through observations and interviews, were analysed in order to draw conclusions that would 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.

As is typical in ethnographic studies, this research went through several stages of gradual refinement. Most of the observations and the interviews in the initial stages of the research,

took place when students were working on their regular learning tasks in the computer laboratory. In these interviews the students were asked to describe what they were doing and how they would explain the behaviour of various functions. Additional observations were made, at that stage, during plenary class discussions. All interviews and class observations were audio taped and transcribed. The analysis of these observations and interviews helped in designing more specific tasks for the second stage of the research. Some of these special tasks involved unfamiliar tasks for the students, while other confronted them with issues that came up during the earlier stage.

There were mainly two kind of tasks: The students were asked either to write a function in the programming language to accomplish a given task, or, given a function in the programming language, to determine what it does or whether it is doing correctly what it was designed to do. The tasks were presented to the students in open interviews as paper-and-pencil tasks, without a computer. The work with paper and pencil focused the students on the task and the procedure at hand, and de-emphasized syntactic issues. It also enabled the researchers to collect for documentation the written materials, including the various drafts. These special tasks were refined several times to help uncover students' conceptions and probe deeper into their understandings.

3.2 Research findings: The clash between the AOS and the function concept

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, 2006). Specifically, we document 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 from the Actions-on-Object Scheme (AOS) clash with the formal function concept. Our data demonstrates two kinds of such clashes: One (discussed in 3.2.1 below), the clash between the AOS intuition of *change* vs. the formalism of function as *correspondence*: Under the influence of the AOS, students perceive the function as changing the input-object into the output-object, yet in the formal function concept nothing really changes – the input-object simply corresponds to the output-object, but itself remains unchanged. Two (discussed in 3.2.2), the clash between the AOS intuition of performing a *chain of actions* on the same (continuously changing) object, vs. the *composition* of functions.

3.2.1. 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 did not appear in the task as given to Dan).

The programming code	Explanation
<i>(define (last-in-second L)</i>	Define a function called <i>last-in-second</i>

with a list L as input variable.

$(\text{cons } (\text{first } L) (\text{cons } (\text{last } L) (\text{rest } L)))$ ¹²

Build a list (going from left to right) from: 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 $\text{rest } L$ 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)].

Dan is correctly trying to trace the evaluation of the expression from right to left.¹³ He is looking at the situation after the execution of $\text{rest } L$ and $\text{last } L$ (which poses no problem) and is trying to figure out the value of $\text{first } L$, specifically, the current value of L that first gets as input. His last pronouncement clearly indicates that he believes that rest has actually chopped off the first element of L . In our terminology, Dan is exhibiting the (implicit) belief that the function rest has changed the value of its input-object 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).*¹⁴

Mili wrote in her notebook $\text{rest } L$ and stopped. Then she called the researcher for help.

Researcher: What's the problem?

Mili: I ask is it possible to do $\text{first } L$ and then I get, like, a ?

Researcher: Is $\text{first } L$ 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. In Dan's words, "When I do *rest L* then the *L* is not (1 2 3 4 5 6) [but (2 3 4 5 6)]". Similarly, Mili, who is discussing the effect of applying *rest* to the list (a b c), says that "I already chopped off the list" and that "it is now left with only *b* and *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 suppresses this view.

It might be claimed that the students in our study could be influenced by their previous experience in the Pascal programming language. However, a more careful examination of its usage, shows that Pascal could have only a minimal effect on AOS-like behaviour. The syntactic candidates from Pascal would be substitutions like $count := count + 1$, which increments the variable *count* by 1, and procedures on strings such as $delete(S,m,n)$, which deletes *n* characters from the string *S* starting from the m^{th} place. However, the former is a substitution, not a function, and the change of variable is mandated explicitly, and the latter, while indeed behaving like the AOS, is used in the course only infrequently. Thus, while there might be some Pascal influence, we believe that it is negligible compared to the influence of the ubiquitous AOS.

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 research subjects learned from. As we have mentioned before, 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”. To get a feeling for the irresistible temptation for teachers (including ourselves) to use AOS-like formulations, readers might try the following thought experiment. Consider the function *F* which inputs a list and a number, and insert the number in the list immediately following the first element of the list which is a number. If you feel that teachers should

avoid using AOS-like formulations (like this one), find a fully formal formulation of the same function F , and see if this is something you would want to use in the classroom.

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", "the selfish gene", or even "imaginary" numbers and "continuous" functions). The rationale for such use is that the metaphor is very useful in the beginning (Lakoff & 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 in depth with the students when the problem actually comes up, drawing their attention to the clash and practicing how to avoid it. This is also the place where forming a rigorous definition of functions like F from the previous paragraph could be a good exercise for the students – not for understanding F , but rather for understanding the clash between the AOS and formal mathematics.

3.2.2. Chaining actions vs. composing functions

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 results in a valid description. But now the question arises,

how do we go from here to the formal programming 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 discouraged (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 (e.g., Ayers, Davis, Dubinsky & Lewin, 1988). The following two cases demonstrate 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 naturally 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 a corresponding one for the last. What one usually does instead is reversing the list and then removing the first element. The remaining actions in the present case would be 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.¹⁵ 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.

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

Sharon has written a precise rendering in the programming language of the above AOS description, but unfortunately it doesn't work. At this point, Sharon called the teacher and asked with puzzlement why the program doesn't work. Our interpretation is that under the influence of the AOS she wrote 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))))*.¹⁶

This demonstrates another facet of the clash between AOS and the formal function concept; that is, the intuition of chaining actions on objects clashes with its formal counterpart of composition of functions. We will return to discuss Sharon's case 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 having been 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 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 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,¹⁷ 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
<code>(define (first-number L)</code>	Define a function called <i>first-number</i> with a list <i>L</i> as input variable.
<code>(cond</code>	Checking the condition:
<code> ((number? (first L)) (first L))</code>	If the first element is a number, output this element
<code> (else (first-number (rest L))))</code>	If not, apply the same function recursively on the list with its first element removed

Part (b), 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
<i>(define (last-number L)</i>	Define a function called <i>last-number</i> with a list <i>L</i> as input variable.
<i>(first-number (reverse L)))</i>	Applying the previous <i>first-number</i> 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. A related phenomenon is reported in Nachlieli and Herbst (2007), whereby students are uncertain as to what kind of mathematical statement they are entitled to use in their proofs.

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 unfortunately to non-trivial technical and conceptual difficulties. It is not easy, for example, to understand why *reverse* should appear *twice* in the last line. Here is one correct version of such a solution:

```
(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, was very agitated and demanded to bring her solution for discussion with the whole class. She 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 again in

translating this idea from natural language to the computer language. As in Sharon's case, we think that this code and the strong conviction that it should work, indicate that Emma was acting under the influence of the AOS. Again, we observe here two AOS-induced problems: chaining actions 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*.¹⁸

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 next, there may be interference between the two kinds of misconceptions due to general cognitive mechanisms.

In two minds

The last example highlights an interesting phenomenon. The participating teachers in that scenario (henceforth called 'the participants') 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 with these participants and their performance on other tasks, we knew that they were 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. The fact that these errors still appeared in their work indicates that the participants' 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 2.3. In terms of dual process theory, we submit that the intuitions of the AOS had initially been part of the participants' System 1 (S1) thinking, since these intuitions are completely automatic and operate below consciousness level. Moreover, they remained part of S1 even after the participants 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 beliefs.) In such situations, where S1 and S2 produce conflicting conclusions, the question arises, which conclusion will the subject finally adopt? Usually, S2 monitors the decisions of S1 and makes the final decision on whether to endorse or override them. However, since S2 relies heavily on the limited resources of working memory (see 2.3), it is less likely to do its S1-monitoring job while being engaged in another complex task, as explained in Kahneman and Frederick (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 a dual process interpretation of the present case, which is rather similar to Kahneman's analysis of the bat-and-ball task in section 2.3. These participants already know that functions do not change their input, but this is S2 knowledge. Usually, when given a task that directly tests the participants' knowledge of whether a function changes its input, the fast automatic S1 response (that it does) may first come to their mind, but is consequently overridden by their S2 knowledge (that it doesn't). In the case reported here, the participants' S2 (with its limited processing capacity) was busy working on the complex recursive task, hence S1 could "hijack" their performance with its quick automatic effortless response, without being "caught" and corrected by the busy S2.

4. Conclusion

In this paper we have investigated one aspect of students' understanding of functions, and the dual role that the intuitive everyday scheme of actions on objects (AOS) can play in this understanding. On the one hand, since such actions have a clearly defined input-state and output-state, and since they can be performed in succession, the AOS can serve as a preliminary, intuitive, embodied basis for learning about functions and their composition (Lakoff & Núñez, 2000; Gray & Tall 2001). On the other hand, we have noted the fundamental conception induced by the AOS, that when we are acting on an object, it undergoes a change but still remains *the same* object. When this fundamental intuition is invoked in the context of functions, it may lead to the erroneously conception that the input-

object is being transformed by the function into the output-object (subsection 3.2.1). Thus, if f is "the function that multiplies its input by 2" and if $x=3$, then students may conclude (incorrectly) that after the application of f to x , x has become 6. We have also presented (in subsection 3.2.2) a related difficulty, that of modelling a *chain of actions* (each operating on the output of its predecessor) by the *composition* of the corresponding functions.

Recognizing these difficulties can help design educational activities to help overcome them. Indeed, we have been using programming activities to create situations where students first encounter these difficulties, and then gradually overcome them. For example, to expose the functions-change-their-input misconception, we may give them a task in which the student has to act twice in succession on the same object (such as the function *last-in-second* from subsection 3.2.1). The student then must come to grips with the question of what has happened to the object after the first operation. This perplexity can be used as a good basis for a classroom discussion of the problem and its resolution. Similarly, to deal with the difficulty with composition of functions, we present students with a complex task (such as *replace-last* in subsection 3.2.2), whose execution requires a succession of actions. The need to translate their verbal solution to the programming language, prompts the students to face the necessity of moving from a series of actions to composition of functions.

To conclude, we have seen that the AOS can serve as an effective intuitive support for learning about functions, but also that it can later clash with the formal function concept, leading to some persistent obstacles in understanding functions. We believe that more research may reveal similar phenomena in other mathematical concepts which have an everyday intuitive support, such as limits and continuity. We suggest that this might be typical of the (cultural) evolution of mathematical concepts: While they may have their origin

in everyday intuitions (Lakoff & Núñez, 2000; Gray & Tall, 2001), 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 striven to suppress all traces of time and process, as demonstrated, for example, by Weierstrass's "discretization program" and the "arithmetization of calculus" (Lakoff & Núñez, 2000, Chapter 14: *Calculus Without Space or Motion*). But our basic intuitions about the world are rooted in action, hence are inseparable from the very same notions (time and process) that are suppressed in the modern formalism.

References

- Abelson, H., & Disessa, A. (1981). *Turtle Geometry The Computer as a Medium for Exploring Mathematics*. Cambridge, Massachusetts: The MIT press.
- Ayers, T., Davis, G., Dubinsky, E., & Lewin, P. (1988). Computer Experiences in Learning Composition of Functions. *Journal for Research in Mathematics Education*, 19 (3), 246-259.
- Bogdan, R. C., & Biklen, S. K. (1998). *Qualitative Research for Education: an Introduction to Theory and Methods* (3th ed.). Mass: Allyn & Bacon.
- Carlson, M., & Oehrtman, M. (2005). Key aspects of knowing and learning the concept of function. *MAA Online*, 9. Retrieved March 15, 2005, from http://www.maa.org/t_and_l/sampler/rs_9.html
- DeMarois, P., & Tall, D. (1999). Function: Organizing Principle or Cognitive Root?. In O. Zaslavsky (Ed.), *Proceedings of the 23th Conference of the International Group for the Psychology of Mathematics Education* (Vol. 2, pp. 257-264). Israel.
- Denzin, N. K., & Lincoln, Y. S. (1994). Introduction Entering the Field of Qualitative Research. In N. K. Denzin & Y. S. Lincoln (Eds.), *Handbook of qualitative Research* (pp 1-17). London: Thousand Oaks. SAGE Publications.
- Dubinsky, E. (1995). ISETL: A programming language for learning mathematics. *Communications in Pure and Applied Mathematics*, 48(9/10), 1027-1052.
- Dubinsky, E., & Harel, G. (1992). The Nature of the Process Conception of Function. In G. Harel & E. Dubinsky (Eds.), *The Concept of Functions: Aspects of epistemology and*

- pedagogy* (Vol. 25, pp. 85-106). Washington, D.C.: Mathematical Association of America.
- Dubinsky, E., & McDonald, M. A. (2002). APOS: A Constructivist Theory of Learning in Undergrad Mathematics Education Research. In D. Holton (Ed.), *The teaching and Learning of Mathematics at University Level: An ICMI Study*. (pp. 275-282). Netherlands: Springer Netherlands Publishers.
- Eisenberg, T. (1991). Functions and Associated Learning Difficulties. In D. Tall (Ed.), *Advanced Mathematical Thinking* (pp. 140-152). Dordrecht/ Boston/ London: Kluwer academic publishers.
- Evans, J.St.B.T (2008). Dual-Processing Accounts of Reasoning, Judgment, and Social Cognition. *Annual Review of Psychology* 59, 6.1–6.24
- Felleisen, M., Findler, R. B., Flatt, M., & Krishnamuethi, S. (2001). *How to Design Programs: An Introduction to Computing and Programming*. Cambridge, Massachusetts: The MIT Press.
- Fischbein, E. (1987). *Intuition in Science and Mathematics: An Educational Approach*. Netherlands: Kluwer Academic Publishers group.
- Geary, D. (2002). Principles of evolutionary educational psychology. *Learning and Individual Differences* 12, 317-345.
- Gigerenzer, G. (2005). I Think, Therefore I Err. *Social Research* 72 (1), 1-24.
- Goetz, J. P., & LeCompte, M. D. (1984). *Ethnography and Qualitative Design in Educational Research*. Orlando, FL: Academic Press.

- Gray, E. M., & Tall, D. O. (2001). Relationships between embodied objects and symbolic procepts: an explanatory theory of success and failure in mathematics. In Marja van den Heuvel-Panhuizen (Ed.), *Proceedings of the 25th Conference of the International Group for the Psychology of Mathematics Education* (Vol. 3, pp. 65-72). Utrecht: The Netherlands.
- Harel, G., & Dubinsky, E. (Eds.). (1992). *The Concept of Functions: Aspects of epistemology and pedagogy* (Vol. 25). Washington, D.C.: Mathematical Association of America.
- 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 T. Frangsmyr (Ed.), *Les Prix Nobel* (pp. 416-499). Retrieved December 28, 2007, from <http://www.nobel.se/economics/laureates/2002/kahnemann-lecture.pdf>.
- Kahneman, D., & Frederick, S. (2005). A Model of Heuristic Judgment. In K. J. Holyoak & R. J. Morrison (Eds.), *The Cambridge Handbook of Thinking and Reasoning* (pp. 267-293). UK: Cambridge University Press.
- Lakoff, G., & Johnson, M. (1980). *Metaphors we live by*. Chicago: The University of Chicago Press.
- Lakoff, G., & Núñez, R. (2000). *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics Into Being*. New York: Basic Books.
- Leinhardt, G., Zaslavsky, O., & Stein, M. K. (1990). Functions, Graphs, and Graphing: Tasks, Learning, and Teaching. *Review of Educational Research*, 60(1), 1-64.

- Leron, U., & Hazzan, O. (2006). The Rationality Debate: Application of Cognitive Psychology to Mathematics Education. *Educational Studies in Mathematics*, 62(2), 105-126.
- Leron, U., & Zazkis, R. (1989). Learning Mathematics through Programming: The Case of Functions and Variables. *Mathematics and computer education*, 3(3), 186-193.
- Nachlieli, T., & Herbst, P. (2007). Students engaged in proving: Participants in the enquiry process of proving or executors of a predetermined script? *Proceedings of the 27th Conference of the International group for the Psychology of Mathematics Education*, 4, 9-16.
- Papert, S. (1980). *Mindstorms: Children, Computers and powerful ideas*. New York: Basic Books.
- Piaget, J. (1983/1970). Piaget's Theory. In P. H. Mussen (Ed.), *Handbook of Child Psychology* (4th ed.) (Vol. 1, pp. 103-128). New York: John Wiley & Sons.
- Ponte, J. P. (1992). The history of the concept of function and some educational implications. *The Mathematics Educator*, 3(2), 3-8.
- Samuels, R., Stich, S., & Tremoulet, P. (1999). Rethinking Rationality: From Bleak Implications to Darwinian Modules. In E. LePore & Z. Pylyshyn (Eds.), *What is Cognitive Science?* (pp.74-120). N.J: Blackwell publishing.
- Schoenfeld, A. H. (1985). *Mathematical Problem Solving*. Orlando, Florida: Academic Press, inc.

- Sierpiska, A. (1992). On understanding the notion of function. In G. Harel, & E. Dubinsky, (Eds.), *The Concept of Functions: Aspects of epistemology and pedagogy* (Vol. 25, pp. 25-58). Washington, D.C.: Mathematical Association of America.
- 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 D. E. Over (Ed.), *Evolution and the Psychology of Thinking: The Debate* (pp. 171-230). UK: Psychology Press.
- Stavy, R., & Tirosh, D. (2000). *How Students (Mis-)Understand Science and Mathematics: Intuitive Rule*. New York: Teachers College Press.
- Stein, E. (1996). *Without Good reason: The Rationality Debate in Philosophy and Cognitive Science*. UK: Oxford University Press.
- Vinner, S. (1983). Concept definition, concept image and the notion of Function. *International journal of Mathematical Education in science and technology*, 14(3), 293-305.
- Vinner, S., & Dreyfus, T. (1989). Images and Definitions for the Concept of Function. *journal for Research in Mathematics Education*, 20 (4), 356-366.

Footnotes

¹ Gray and Tall (2001) point to important differences between the two theoretical frameworks, especially their use of ‘embodiment’; however, these differences will not affect the present argument.

² For our purposes it is useful to consider the embodied objects together with the actions (also embodied) which they naturally afford, hence the ‘embodied scheme’.

³ Thanks to Lisser Rye Ejersbo.

⁴ “An action is a transformation of objects perceived by the individual as essentially external and as requiring, either explicitly or from memory, step-by-step instructions on how to perform the operation” (Dubinsky & McDonald, 2002, p. 276).

⁵ 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.

⁶ “When the symbols act freely as cues to switch between mental concepts to think about and processes to carry out operations, they are called procepts.” (Gray & Tall, 2001, pp. 67-68)

⁷ Tversky unfortunately died several years earlier.

⁸ Note the slight difference in syntax: In Scheme we write (*first L*) instead of the familiar mathematical notation *first(L)*.

⁹ In CS there are some extreme cases of functions that appear to have no input or no output, but we will ignore them here.

¹⁰ DrScheme has been developed by the PLT Scheme group (<http://www.plt-scheme.org/>), with the objective of offering a pedagogical environment for functional programming (Felleisen, Findler, Flatt & Krishnamuethi, 2001). The software can be freely downloaded from <http://download.plt-scheme.org/drscheme/>.

¹¹ The students' prior experience in procedural programming included variables, conditional statements, loops, one-dimensional arrays, but not functions.

¹² In standard mathematical notation: $\text{cons}(\text{first}(L), \text{cons}(\text{last}(L), \text{rest}(L)))$.

¹³ The Scheme interpreter parses expressions from left to right (top-down), but students usually check execution from right to left (bottom-up).

¹⁴ Typical solution: $(\text{cons } x (\text{cons } \text{first } L (\text{cons } x \text{ rest } L)))$,
or in mathematical notation, $\text{cons}(x, \text{cons}(\text{first}(L), \text{cons}(x, \text{rest}(L))))$.

¹⁵ Concrete is a relative term, and a list of numbers or words is quite concrete for these students.

¹⁶ In mathematical notation: $\text{reverse}(\text{cons}(n, (\text{rest}(\text{reverse}(L)))))$.

¹⁷ Standard too is a relative term, and this kind of recursive idiom has become quite standard for these students.

¹⁸ The precise details may differ between different functional dialects, but these differences do not affect the present analysis.