A case of teaching practice founded on a theoretical model

Sylvia da Rosa, Marcos Viera, and Juan García-Garland

Instituto de Computación, Facultad de Ingeniería, Universidad de la República.
{darosa,mviera,jpgarcia}@fing.edu.uy

Abstract. This paper tries to clarify the way in which our theoretical model relates to teaching practice in response to questions about how the model could potentially be applied. The theoretical model introduces an extension of Jean Piaget's general law of cognition to explaining the difference between algorithmic thinking and computational thinking by adequately locating the latter in the specificities of the subject instructing a computer. The teaching practice consists on activities introducing programming in high school mathematics courses. These are organised in a functional programming course to high school mathematics teachers and didactic instances in which the teachers teach their students to program solutions to mathematics problems.
Through examples we explain how the model helps teachers in finding a meaning of the popular and controversial expression "computational thinking". The goal of the didactic instances is to educate students in thinking algorithmically and computationally.

1 Introduction

This paper tries to clarify the way in which a theoretical model relates to teaching practice in order to response to question of how the model could potentially be applied.

The theoretical model introduces an extension of Jean Piaget's general law of cognition, that explains how conceptual knowledge is constructed when an individual solves a problem [10]. We have applied Piaget's general law of cognition to investigate the construction of knowledge of algorithms and data structures. In the case of knowledge of programs, the research led us to extend Piaget's general law of cognition, because the goal is to know how to help students learning *how to program*, not just to know how to help them learning *how to write program texts* (algorithms). The ontological approach of our programming didactics considers that a program is in some sense a synthesis between a text (an algorithm) and a machine that executes it. That means that knowledge about the text becomes necessary but not sufficient to deal with programming problems. The research of the construction of knowledge about programs has two main results: it offers a theoretical model that explains the relationship between conceptual knowledge of algorithms and of programs [13], and gives a clear definition of the controversial expression "Computational Thinking" (hereinafter CT) [12]. In this paper

we describe how and why teachers activities introducing programming in high school mathematics courses are an example of the application of the theoretical model. At the same time, the experience describes a way of introducing CT into school, contributing to educators' concern since CT became popular in educational settings [5, 9].

The rest of the paper is organised into the following sections: in Section 2 we describe the theoretical model, while in Subsection 2.1 we introduce the extended law of general cognition; in Section 3 we describe a teaching experience with mathematics teachers and explain how and why it relates to the theoretical model, and contributes to clarify teachers and students ideas of CT. Finally, we include conclusions and references.

2 The theoretical model

In Piaget's theory, human knowledge is considered essentially active, that is, knowing means acting on objects and reality, and constructing a system of transformations that can be carried out on or with them [11].

The problem of determining the role of experience and operational structures of the individual in the development of knowledge **before** the formalisation was studied in depth by Piaget in his experiments about genetic psychology. From these he formulated a *general law of cognition* [10], governing the relationship between know-how and conceptualisation, generated in the interaction between the subject and the objects that he/she has to deal with to solve problems or perform tasks. It is a dialectic relationship, in which sometimes the action guides the thought, and sometimes the thought guides the actions.

Piaget represented the general law of cognition by the following diagram:

$$\text{C} \leftarrow \text{P} \rightarrow \text{C'}$$

where P represents the periphery, that is to say, the more immediate and exterior reaction of the subject confronting the objects to solve a problem or perform a task. This reaction is associated to pursuing a goal and achieving results, without awareness neither of actions nor of the reasons for success or failure. The arrows represent the internal mechanism of the (algorithmic) thinking process. By that process the subject becomes aware of the coordination of the actions -a method- that she/he has employed to solve the problem (P $\rightarrow$ C in the diagram) and of the modifications that these actions impose to objects, as well as of objects' intrinsic properties (P $\rightarrow$ C' in the diagram). C and C' represent awareness of the actions (maybe mental) encapsulated in the algorithm and of the data structures, respectively. The process of the grasp of consciousness described by the general law of cognition constitutes a first step towards the construction of concepts.

2.1 The extended law of general cognition

The construction of knowledge about algorithms and data structures is a process regulated by the general law of cognition. Over the years we have investigated the

construction of knowledge by novice learners of algorithms and data structures (for instance sorting, counting, searching elements) basically by applying Piaget's law. A synthesis of previous work can be found in [14].

However, in the cases where the subject must instruct an action to a computer, the thought processes and methods involved in such cases differ from those in which the subject instructs another subject, or performs the action him/herself. In order to program a computer to solve a problem, the learners have to establish a causal relationship between the algorithm (he/she acting on objects), and the elements relevant to the execution of the program (the computer acting on states). By way of analogy with Piaget's law we describe that causal relationship by the following diagram:

$$\begin{array}{ccccc} & & \underbrace{C & \leftarrow & P & \rightarrow & C'} & & \\ newC & \longleftarrow & newP & \longrightarrow & newC' \end{array}$$

where newP is characterised by a periphery centred on the actions of the subject and the objects he/she acts on. The centres newC and newC' represent awareness of what happens inside the computer: newC of the execution of the program instructions and newC' of the undergone modifications of the representation of data structures. The causal relationship between the first row and the second row is the key of the knowledge of a machine executing a program. It is indicated with the brace in the diagram above. The diagram describes the situation in which the subject reflecting on his/her role as problem solver becomes aware of how to do to make the computer solve the problem [16].

According to Piaget, we identify that the construction of knowledge of methods (algorithms) and objects (data structures) occurs in the interaction between C, P and C'. Likewise, we claim that the construction of knowledge of *a program as an executable object* takes place in the internal mechanisms of the thinking process; marked by the arrows between newC, newP and newC'. In other words, the general law of cognition remains applicable to the thinking process represented by the arrows; in both lines of the diagram pictured above. In [13] we describe an empirical study in which we introduce the extended law of cognition (hereinafter extended law). In [12] we explain that our extension of Piaget's law introduces a clear definition of the notion of CT (represented by the second line of the above diagram). Further, this new definition is adequately located in relation to the notion of algorithmic thinking (represented by the first line of the above diagram). The above diagram represents the theoretical model of the construction of knowledge of algorithms, data structures and programs.

In the next section we describe a teaching experience and explain how and why it relates to the theoretical model and contributes to clarify ideas of CT in educational settings. It consists of activities that provide teachers with a clear description of CT according to our theoretical definition. The teachers become trained to help the students learning to program, in a way that respects the process of learning how to think algorithmically and computationally.

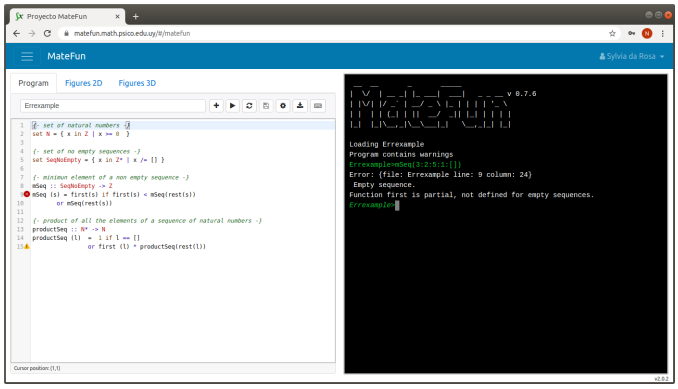

Fig. 1: MateFun IDE

3 CT in educational settings

The activities were developed in 2019 and consisted of a six weeks programming course for high school mathematics teachers and the supervision of six weeks activities that these teachers carried out with their students after taking the course. The teachers teach in diverse educational centres in different regions of Uruguay. They also teach in different high school years, and most of them teach to more than one group of students.

The main objective of the activities is the integration of mathematics and programming as a way to facilitate understanding and learning of mathematical concepts and at the same time to provide basic programming knowledge to high school students (see [15] for more details).

The guiding idea of the experience comes from [8] (page 327), in the sense that programming a solution to a problem exposes aspects of the resolution process that are otherwise hidden. Some examples are the need of formulating algorithmic problems as such; the role of programming for the training of abstraction; the possibility of comparing algorithms and choosing those that are more efficient; and the introduction of a method of proof of equivalence between algorithms (see Section 3.2, first, second, third and fourth example respectively).

The functional language MateFun, briefly described in the next subsection, was used in the experience (see [15] for more details).

3.1 The language MateFun

The language MateFun [3,4] is a functional programming language designed with the specific purpose of being a tool to support mathematical learning, especially in high school.

MateFun is purely functional, meaning that functions do not introduce side effects and they only depend on their arguments. To be easily approachable it

is available as a web integrated development environment (Matefun IDE[1]), as shown in Figure 1. The left frame is a text editor, where the program is written, and the right frame is a shell with a read-eval-print-loop, where the programs can be loaded (if they are correct) and the expressions evaluated.

Syntax and semantics of MateFun are both influenced by the seek to be a tool to express mathematics. The syntax is minimal and close to the usual mathematical notation. Semantically, it has the peculiarity of being strongly typed, while having no type inference. The skill to specify the domain and range of a function is part of the learning process when learning about functions. In MateFun type information must be given by users, and types in MateFun are called sets.

A MateFun script is a list of definitions of *sets* and *functions* over such sets. Predefined sets such as `R` (representing real numbers) or `Z` (representing integer numbers) are available as built-in constructs.

The user can define new sets either by comprehension or by extension, just as usually presented in mathematics courses. In the following example we define the sets of natural numbers (`N`), non-zero real numbers (`Rno0`), days of the week (`Day`) and (`Bool`):

```
1 set N    = { x in Z | x >= 0 }
2 set Rno0 = { x in R | x /= 0 }
3 set Day  = { Mon, Tue, Wed, Thu, Fri, Sat, Sun }
4 set Bool = {True, False}
```

Sets such as `Rno0`, defined by comprehension, take a base set (`R` in this case) and refine it with a predicate. Predicates can be built by relational operators and from other predicates by conjunctions.

Functions are defined giving a signature and a proper definition. For example, one could define the inverse function over the non-null real numbers:

```
5 inv :: Rno0 -> R
6 inv (x) = 1/x
```

MateFun supports some of the idioms used to define functions in mathematics. For instance, piece-wise functions can be defined, while, unlike most functional languages, it does not support pattern matching or conditional expressions. The following MateFun definition specifies the absolute value function over the real numbers:

```
7 abs :: R -> R
8 abs (x) =   x if x >= 0
9         or -x
```

This program resembles the definition in the usual mathematical notation:

$$abs : \mathbb{R} \to \mathbb{R}$$

$$abs(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

[1] `https://matefun.math.psico.edu.uy`

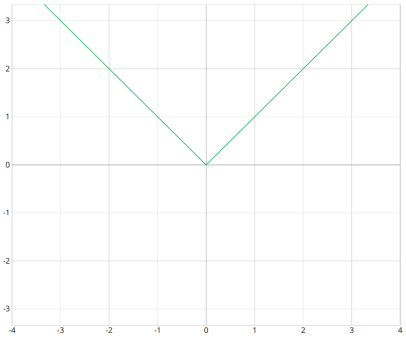

Fig. 2: Function plot: `?plot abs`

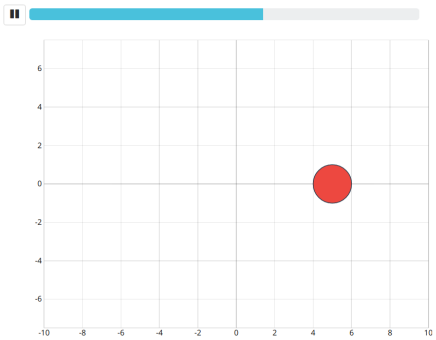

Fig. 3: `moveRight(redCirc(1),10)`

Then, if we load the program in the interpreter, we can ask to compute the absolute value of the number `-10` by typing the expression:

```
Example>abs(-10)
10
```

The interpreter evaluates expressions and interprets special commands. For instance, we can graph a (one-variable) function using the command `?plot`. The result of plotting `abs` is shown in Figure 2.

```
Example>?plot abs
```

To emphasise that not all functions are numeric, MateFun allows to define non-numeric sets (eg. `Day` and `Bool`) and functions between those sets, such as `holiDay`:

```
10 holiDay :: Day -> Bool
11 holiDay (d) =  True if  d == Sun
12              or True if  d == Sat
13              or False
```

We can also define *sequences* of elements of a given set; usually called *lists* in programming. The sequence set `A*` is defined inductively as:

- `[]`, the empty sequence
- `a:as`, a sequence composed by an element `a` belonging to `A` and a sequence `as` belonging to `A*`.

For instance, `N*` is the set of sequences of natural numbers.

There exist some primitive functions to operate with sequences: `first(s)` returns the first element of the sequence `s`, `rest(s)` returns the sequence `s` without its first element, and `range(n,m,k)` returns a sequence of numbers $(n, n+k, n+2k, ...)$ from `n` to `m` with step `k`. With `range`, combined with a function to `sum` the elements of a sequence, we can for instance implement the summatory $\sum_{i=m}^{n} i$.

```
14 summatory :: N X N -> N
15 summatory (m, n) = sum(range(m, n, 1))
16
17 sum :: R* -> R
18 sum (xs) =  0 if xs == []
19           or first(xs) + sum(rest(xs))
```

Notice the use of recursion in the implementation of `sum` and the domain using two variables. Domains with multiple variables can be defined using *n-tuples* (the generalisation of Cartesian products).

The language includes the primitive sets `Figure` and `Color`, and a set of primitive functions to create and transform *figures*. For example, the following function returns a red-coloured circle of a given radius, centred in the $(0, 0)$ point of a Cartesian plane.

```
20 redCirc :: R -> Fig
21 redCirc (r) = color(circ(r), Red)
```

In MateFun, *animations* are sequences of figures. The following function takes a figure `fig` and a number `n` of steps, and returns an animation in which the figure is moved `n` times one step to the right on the x-axis:

```
22 moveRight :: Fig X Z -> Fig*
23 moveRight (fig, n)
24   =  [] if n == 0
25   or fig : moveRight(move(fig, (1, 0)), n - 1)
```

The expression `move(fig, (1, 0))` moves `fig` to the point obtained adding `1` and `0` to the abscissa and the ordinate of the centre of `fig` respectively. Figure 3 shows the sixth frame of the following animation:

```
Example>moveRight(redCirc(1),10)
```

3.2 Teachers and students activities

The first part of the experience is a MateFun programming course to mathematics teachers and the second part consists of activities that teachers do with their students. A complete description of teachers and students activities can be found in [15], as well as teachers' reasons of their choices of problems and comments about students work. Here some activities are selected to describe how those relate to our theoretical model and offer a way of introducing CT in classrooms.

In educational settings, the starting point for developing a program is usually the design of the program's text, that is, an algorithm. In other words, the first step in getting the computer to solve a problem is to have an algorithm that allows an individual to solve specific cases of the problem. The construction of knowledge in this first step is regulated by Piaget's general law of cognition (see Section 2). Many of the problems covered in the course, are contributed by the teachers themselves and they know the solutions. In the course they are trained to program these solutions in MateFun, that is, to teach a computer to solve

the problems, a process regulated by the extended law (see Section 2.1). Facing this challenge teachers' thinking starts from the new periphery (newP), since they manage to elaborate an algorithm to solve the problem, and it has to move towards the elements of program execution (newC and newC'). However, the process is dialectical and often, the transition of the thought from the algorithm and data structures, (C and C') to elements of program execution (newC and newC'), shows the need to modify the algorithm, and even the formulation of the problem. In the examples below, part of that dialectical process is described.

First example: computability A theme introduced by the teachers in a group of third-year high school students (aged 13-14) is related to the problem of finding the multiples of a natural number. The accumulated experience[2] has taught us that often mathematics teachers are not used to rigorously formulate problems; many times they formulate problems forgetting details that are unconsciously interpreted, as in this case. The result could be expressed as "0, 3, 6, 9, ..." or "0, 3, 6, 9, and so on". However, if the question is to write the solution in a rigorous language, such as a programming language, and executing the program, neither the dots nor "and so on" are acceptable. That means that the problem has to be reformulated in terms of input-output [7], to explicitly including the natural number *and a bound* as input. This is an example of going from algorithmic to computational thinking, or in terms of the extended law, from newP to newC and newC', because thinking on program execution requires new knowledge about the input (newC') and the actions (newC).

It is worth mentioning that MateFun is a more powerful tool than other languages (Python, C), since being strongly typed, it requires to write the signature of the function as part of its definition (in this case, forcing to express the input as a pair of natural numbers). The solution in MateFun can be found in [15].

Simple examples like this give teachers the opportunity to introduce computing problems such as computability, in an understandable way by the students.

Second example: abstraction Although a solution of a general problem has to be expressed as an algorithm, the power of abstracting is revealed in all its magnitude when the algorithm is transformed into a program that can be executed for several cases. The process has a greater impact in the case of novice learners, for whom the possibilities of experimenting their solutions in action is a reason of high motivation, as teachers comment in [15]. We illustrate the case using the following example.

Teachers asked the students to program a function that makes a circle move n steps through the points of multiples of three on the x axis. A solution is presented below, in which the students adapted the function `moveRight` (see 3.1) to program a function `move3` that moves a figure n steps through points corresponding to multiples of three on the x axis. They also programmed `move3cir` in which the parameter `fig` of `move3` is instantiated to a red circle previously

[2] We have taught the course for about twelve years using other languages [15].

defined. Notice that adapting `moveRight` induced the students to write `move3`, abstracting the figure in the parameter `fig` as in `moveRight`.

```
circle :: R X Color -> Fig
circle (r, c) = color(circ(r), c)

move3 :: Fig X Z -> Fig*
move3(fig,n) =  [] if n < 0
             or fig : move3(move(fig, (3, 0)), n-1)

move3cir :: N -> Fig*
move3cir (n) = move3(circle(0.5, Red), n-1)
```

Since the problem asks for the multiples of three, the students used the concrete case of the point `(3,0)` in the function `move3`. When the teachers presented this solution in our course they were asked how to generalise it to the multiples of *any natural number*, that is, abstracting the point (3,0) to `(num,0)`. To construct a general solution they observed that `move(fig,(num, 0))` moves the figure `fig` through the multiples of any natural number -represented by `num`- on x axis. The point discussed was how to define a single function that also performs the movement n times. Observe that in terms of the general law of cognition that means that the thought advances towards newC' (the parameters) and newC (the action of moving), respectively. Finally, teachers introduced the definition of `moveMultiples` below.

```
moveMultiples :: Fig X Z X N -> Fig*
moveMultiples(fig, n, num)
  =  [] if n == 0
  or fig : moveMultiples (move(fig,(num, 0)),n-1, num)
```

For instance, moving the red circle through the multiples of three, five times, is obtained with `moveMultiples(circle(0.5, Red), 5, 3)`.

The point is to always start from teachers or students solutions to construct new ones. In this type of exercises not only the abstraction is trained but also the skill of getting better programs by composing and combining other functions (predefined or not).

Third example: complexity In our course the teachers are asked to solve many problems involving programming of functions over sequences, using recursion and/or composition of functions. The example below shows a MateFun program for the factorial function (`fact`). This is a well known definition by the teachers and all of them succeed in solving this problem.

```
fact :: N -> N
fact (n) =  1 if n == 0
         or n * fact(n-1)
```

Then the teachers are asked to write another program (`factorial`) for the same function using two functions: `productSeq` that returns the product of the elements of a sequence (see below), and `range` introduced in Section 3.1.

```
1 productSeq :: Z* -> Z
2 productSeq (l)  =  1 if l == []
3                    or first (l) * productSeq(rest(l))
```

Most of the teachers arrived to a solution similar to:

```
1 factorial :: N -> N
2 factorial (n) =  1 if n == 0
3                  or productSeq(range(1, n, 1))
```

It can be observed that the definition has a redundant equation in line 2, induced by the recursive definition of `fact`. The equation in line 2 is used for a case that is actually encompassed by the definitions of functions `range` (case b < a) and `productSeq` (`productSeq([]) = 1`). This kind of errors in which edge cases are mishandled are frequently made by both teachers and students. Several lessons are learnt from solving that kind of exercises. One of the most important is that a program perhaps gives the correct result (teachers' solution of `factorial` does), but has errors anyway. From the point of view of programming it is an error to make the computer do things that are not necessary or are redundant. Understanding why the redundant equation is an extra effort for a computer is a clear example of transiting from algorithmic to computational thinking. In terms of the the general law of cognition that means understanding how the computer performs the actions (newC) on the objects (newC'). One could argue that the redundant equation can be noted even in the algorithm and this is true, especially to more experienced programmers. In early stages of learning, however, to experience the need of a concept is the first step of constructing the concept [10] (in this case the need of efficiency). This training gives teachers the opportunity of introducing their students into topics such as programs complexity, inducing them to think computationally.

Fourth example: program correctness Programming more than one solution to a problem gives us the opportunity of introducing teachers to a topic that brings mathematics and functional programming even closer: the use of the principle of *structural induction* to prove properties of programs. For instance, one can prove that the two above definitions of the function factorial are equivalent in the sense that both give the same result when applied to the same value. The property is:

Property 1. $\forall$ `n` $\in$ `N`, `fact(n) = factorial(n)`.

and is proved using the principle of structural induction. For space reasons it is not included here, but can be found in [15].

Although the proofs are done with pen and paper, it is possible to do them using equational reasoning (substituting equals for equals where every step has to be well founded) since MateFun is a pure functional language (without side effects).

It is worth saying that traditionally, the principle of induction is used in a very restricted way in high school education, usually making no sense for students.

Teachers could teach this method as a way of verifying program correctness, in other words, thinking about the correctness of the computer's actions (newC) on the objects (newC'). At the same time, the students would learn the basis of a topic of mayor relevance for understanding computer science.

4 Conclusions

The absence of clear definitions and substantiated claims about CT, "... leave teachers in the awkward position of not knowing exactly what they supposed to teach or how to assess whether they are successful". Those are P.Denning's words in [5]. In fact, the application of a concept that is theoretically weak can even be counterproductive. As L.Paulson notes in [9]: "Unless somebody can come up with a more insightful definition, it is indeed time to retire 'computational thinking'".

Taking principles of Jean Piaget's theory, Genetic Epistemology [12], our theoretical model offers an insightful definition for CT adequately located in relation to the notion of algorithmic thinking. Therefore, our definition leaves teachers in a position of knowing ideas of CT in educational settings and being able to decide how to apply them.

Our claim is that any learning process is built stepwise and is governed by the general law of cognition. In the specific case of learning to program, the process is governed by the new law of cognition as we have formulated it on Section 2.1. Teachers activities presented in this paper show how our theoretical model of CT relates to teaching practices. Particularly, these activities help teachers to understand what CT means and how introduce the students in learning to program, in a way that respects their teaching practices. As a consequence teachers and students are educated to think algorithmically and computationally.

Furthermore, our contribution satisfies Denning and Matti Tedre definition (chapter 1 of [6]) in the sense that not only the activities show how "to get computers to do jobs for us", but introduce teachers and students in some of the relevant problems of computer science, that make "the world a complex of information processes" [6]. For instance, the examples show how computability and complexity of programs could be discussed at high school level. The examples also reveal the power of abstraction in all its magnitude when putting into practice one of the main contributions of CT: to make the computer to solve general problems. Abstracting from concrete cases to obtain generic elements is not a trivial issue, and learning to program plays a fundamental role in training of abstraction from early stages, as several authors indicate [1, 2, 16, 17].

Our theoretical model explains -in the framework of Piaget's theory of construction of knowledge- the relationship between logic, definitions, properties and proofs (algorithmic thinking), and the elaboration and execution of programs (computational thinking). The presented examples constitute examples of a didactic application of the model insofar as they show how the model supports teaching practice.

References

1. Aho, A.V.: Computation and Computational Thinking. The Computer Journal **55** (2012)
2. Ambrosio, A.P., da Silva, L., Macedo, J., Franco, A.: Exploring Core Cognitive Skills of Computational Thinking. Proceedings of the 25th Annual Psychology of Programming Interest Group Workshop, University of Sussex (2014)
3. Cameto, G., Carboni, A., Koleszar, V., Méndez, M., Tejera, G., Viera, M., Wagner, J.: Using functional programming to promote math learning. In: 2019 XIV Latin American Conference on Learning Technologies (LACLO). pp. 306–313 (2019)
4. Carboni, A., Koleszar, V., Tejera, G., Viera, M., Wagner, J.: Matefun: Functional programming and math with adolescents. In: Conferencia Latinoamericana de Informática (CLEI 2018) - SIESC (2018)
5. Denning, P.J.: Remaining Trouble Spots with Computational Thinking. Communications of the ACM **60** (2017)
6. Denning, P.J., Tedre, M.: Computational Thinking. The MIT Press, Cambridge, Massachusetts, London, England (2019)
7. Harel, D., Feldman, Y.: Algorithmics The Spirit of Computing. Addison-Wesley. An imprint of Pearson Education Limited (2004)
8. Knuth, D.: Computer Science and its relation to mathematics. Basic Books, Inc., Publishers / New York (1974)
9. Paulson, L.C.: Computational Thinking is not Necessarily Computational. Communications of the ACM **60** (2017)
10. Piaget, J.: La Prise de Conscience. Presses Universitaires de France (1964)
11. Piaget, J.: Genetic Epistemology, a series of lectures delivered by Piaget at Columbia University, translated by Eleanor Duckworth. Columbia University Press (1977)
12. da Rosa, S.: Piaget and Computational Thinking. CSERC '18: Proceedings of the 7th Computer Science Education Research Conference pp. 44–50 (2018)
13. da Rosa, S., Aguirre, A.: Students teach a computer how to play a game. LNCS 11169: 11th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2018 pp. 55–67 (2018)
14. da Rosa, S., Gómez, F.: Towards a research model in programming didactics. Proceedings of 2019 XLV Latin American Computing Conference (CLEI) p. 1–8 (2019). https://doi.org/10.1109/CLEI47609.2019
15. da Rosa, S., Viera, M., García-Garland, J.: Mathematics and MateFun, a natural way to introduce programming into school. `https://hdl.handle.net/20.500.12008/25233` (Last accessed September 2020)
16. Seymour Papert: Papert, S. Mindstorms: Children, Computers, and Powerful Ideas. Basic Books (1980)
17. Wing, J.: Computational thinking and thinking about computing. Philosophical transitions of the Royal Society **Phil. Trans. R. Soc. A 366**, 3717–3725 (2008)