The Waouh Project

Félix Ridoux

Université de Rennes 1

June 2020

Abstract

The Waouh project is to design an execution environment for WHILE programs to make easier the learning of iterative programming for beginners. The global idea is to improve the experience of the user by allowing step-by-step execution and visualization of the memory, and reverse execution to help understanding causes and effects in iterative programming. The Waouh project is developed as a web application.

1 Introduction

The Waouh project is rooted in our beginner programmer experience and the observation that many people have difficulties to learn the very first principles of iterative programming. We believe the difficulty is aggravated by two factors:

- Using an actual programming language which blurs elementary concepts into non-elementary constructions.
- Using an actual integrated development environment (IDE) which blurs even more elementary concepts.

However, beginner programmers need not know what an Eclipse project is when their very difficulty is to understand the difference between expressions and statements, between the left occurrence of a variable in an assignment and its right occurrence, or between the WHILE loop and the FOR loop. These concepts should be presented as pure as possible to the beginner, and in a way that makes experiments very easy.

The mission of the Waouh project is to offer this service. So, it will not be driven by software engineering issues, neither by efficiency issues, neither by expressivity issues. It is only designed to help playing with the basic concepts, testing ideas, and running What-If experiments. What happens if I change the loop condition? What happens if I permute these two statements? Etc. There should be the least interface possible between the beginner programmer and the programming concepts.

2 The Waouh system

The Waouh system is an answer to the previous motivations. It is the result of three main design decisions: an iterative programming language, an interaction model, and an interface.

2.1 An iterative programming language

2.1.1 Principles

The iterative programming language that we have chosen for the Waouh system is a variant of the WHILE programming language which is very close to the variant in [1]. It is also very close to the language used in our computability theory course at university. It offers classical constructions like sequence, IF construct, WHILE loop, and FOR loop. It does not aim at being the minimal programming language. For instance, it offers both WHILE loops and FOR loops, even if in theory WHILE loops suffice. But if there where only WHILE loops, it would be impossible to experiment the difference between WHILE loops and FOR loops. Similarly, the IF construct could be encoded in contrived WHILE loops, but hiding IF statements is certainly not the best approach to play with them.

In the same spirit, we did not adopt minimalistic variants of the WHILE language where values are only integers [2]. Indeed, though it is theoretically possible to code everything using integers, it does not really correspond to common programming. Instead, we adopt the variant of Neil Jones's book [1] where values are LISP-like consed trees. We cite Neil Jones preface (emphases are from the source):

> *I introduce a simple imperative programming language called WHILE, in essence a small subset of Pascal or LISP. The WHILE language seems to have just the right mix of expressive power and simplicity.* Expressive power *is important when dealing with programs as data objects. The data structures of WHILE are particularly well suited to this, since they avoid the need for nearly all the technically messy tasks of assigning* Gödel numbers *to encode program texts and fragments (used in most if not all earlier texts), and of devising code to build and decompose Gödel numbers.* Simplicity *is also essential to prove theorems about programs and their behavior. This rules out the use of larger, more powerful languages, since proofs about them would be too complex to be easily understood.*

Our goal is not to prove theorems, but we believe this choice applies as well to learning programming concepts. Using consed trees, it is really easy to play with basic data structures like list and trees, and even to start with beginner algorithmics. A strong limit is that these structures are immutable, though many algorithms use mutable structures. As pure functional programmers know an immutable variant can often be found, but it is no longer a beginner problem.

2.1.2 Specification

The specification of WHILE can be presented in cleanly separated layers. This is not always so clean with real programming languages, but we believe that learning to program is not only learning to code in a given language, but it is also learning how programming languages are defined.

The value layer First layer of specification is the value domain of WHILE programs. It is the domain of *binary trees*. Binary trees can be *atoms*, denoted by symbol identifiers, the *empty tree*, denoted by $\otimes$, and *consed trees*, denoted by $l \bullet r$, where l and t are binary trees.

There is no type in WHILE; more precisely there is only one type, binary tree, hence no type differences. However, familiar types can easily be emulated using binary trees. For instance, booleans can be coded by consed trees for `true`, and non-consed trees for `false`. Integers can be coded as follows. Zero is coded as a non-consed tree like $\otimes$, and $n+1$ is coded as $t \bullet$ *code-of-*n, where t is any binary tree. Conversely, all WHILE values can be interpreted as booleans or integers by following these rules in reverse. For instance, $t \bullet u \bullet v \bullet w$ can be interpreted as `true` or 3 $(0+1+1+1)$, assuming w is a non-consed tree. With this encoding, many different trees can represent the same integer or the same boolean value, and every binary tree can be interpreted as a unique integer and a unique boolean value.

Values are *assigned* to *variables*. Since there is no declaration in WHILE, a convention is that variable identifiers always start with a capital letter, to differentiate them from symbol identifiers which always start with a small letter. What value is assigned to a variable is recorded in a *memory*:

$$mem : variable \longrightarrow binary\ tree.$$

The expression layer Next layer is *expressions*. Expressions denote values (or evaluates to), either directly, or via a variable and a memory contents, or via operations on binary trees. `nil` denotes $\otimes$, and if `l` denotes l and `r` denotes r then `(cons l r)` denotes $l \bullet r$. Only three operations are defined, `hd` (head) and `tl` (tail), and `=?`.

- If `t` evaluates to $l \bullet r$, then `(hd t)` evaluates to l, and `(tl t)` evaluates to r. All other cases evaluates to $\otimes$. So, `hd` and `tl` are the left and right accessors of binary trees. They are defined for all binary trees.

- If `l` and `r` evaluate to the same value, then `l =? r` evaluates to a consed tree (i.e. `true`, see the value layer), otherwise it evaluates to a non-consed tree (i.e. `false`). So, `=?` is the comparator of binary trees. Which consed tree represents `true` and which non-consed tree represents `false` is not defined, but it is stable so that all `true` values produced by `=?` are the same. Similarly for `false` values.

Variables denotes what the memory contents records have been assigned to them: *V* denotes $mem(V)$. The default is $\otimes$. Expressions must always be evaluated with respect to some memory contents.

It is important that learners understand that expressions are not values; they denote values. In particular, there are variables in expressions, but not in values. It is also important to understand that evaluating an expression never changes the memory contents. As opposed to constructs like `++i` or `i++` that are familiar in many programming languages but mask fundamental concepts with superficial ones like pre- or post-indexation.

The command layer Next layer is *commands* (also known as statements, or instructions). They are used to change the memory contents as follows:

- `nop`: leaves the memory contents unchanged.
- *command* `;` *command'*: executes *command*, then *command'*. The memory contents is changed by *command* and then by *command'*. *command'* starts with the memory contents as *command* left it.
- *variable* `:=` *expression*: records that the value assigned to *variable* is the value of *expression*. Assignment commands and the `nop` command form the category of *simple commands*. Other commands are called *complex commands*.

  Since it is a value that is assigned to a variable, and not an expression, there is no ways that an assignment may change the value of a variable other than the variable in left position. There is no hidden side-effect. All that changes is what is indicated to change.

```
A := nil ;              % A represents 0
B := (cons nil A) ;     % B represents 1
A := (cons nil A).      % A represents 1
                        % but B still represents 1
```

- `if` *expression* `then` *command* `else` *command'* `fi`: evaluates *expression*, interprets it as a boolean b (see value layer above), executes *command* if b is `true`, executes *command'* otherwise.

  command is called the *then branch* of the condition, *command'* its *else branch*, and *expression* its *condition*.

- `while` *expression* `do` *command* `od`: evaluates *expression*, interprets it as a boolean b (see value layer above), executes *command* if b is `true`, and repeats.

  command is called the *body* of the loop, *expression* its *condition*.

  The learner must understand that a WHILE loop may run into an *infinite loop*, as in:

```
while (cons nil nil) do nop od
```

In short, the role of the loop body is to make the loop condition become `false`.

- `for` *`expression`* `do` *`command`* `od`: evaluates *`expression`*, interprets it as an integer n (see value layer above), and starts n turns of executing *`command`*.

  `command` is called the *body* of the loop, *`expression`* its *counter*.

  It is important that learners understand that FOR loops always execute a finite number of turns. This is the case even if the body of the loop seems to alter the count of the loop, as in:

  ```
  for X do X := (cons nil X) od
  ```

  where `X := (cons nil X)` seems to alter the count of the loop. The counter of the FOR loop is evaluated once when entering the loop, and this alone decides the number of turns. The loop body plays no role in deciding the number of turns.

  In many programming languages, FOR loops can be forced to behave as WHILE loops, so that they loose their always-terminating property. This makes the difference between WHILE loops and FOR loops unclear.

The program layer Finally, the last layer is *programs*. Programs are connected to a calling environment that gives inputs and expects outputs:

`read` *`inputs`* `%` *`command`* `%` `write` *`outputs`*`.`

They read *inputs* and write *outputs*. They may have several inputs and outputs, as in:

```
read A, B % command % write U, V.
```

Inputs are variables that get there initial values from the calling environment. Apart from that they behave as ordinary variables, and can be assigned to new values if needed. Similarly, outputs are ordinary variables, except that their final values are communicated to the calling environment.

For instance, the following program computes the sum and product of two numbers:

```
read A, B
%
    S := A ; P := nil ;
    for B do
        S := (cons nil S) ;
        for A do P := (cons nil P) od
    od
%
write S, P
```

2.2 An interaction model

2.2.1 Principles

The interaction model proposed by the Waouh system is basically step-by-step execution with visualization of the memory after each step. To overcome the need to tediously step through a program execution from its very beginning to some point of interest, the jump-to-breakpoint variant is also proposed.

This simple model of interaction is quite popular in environment for animating algorithms. However, it has been shown that this is not enough to improve learning [5]. This article shows that it is necessary to give to the learner some means to explore the program execution in such a way that all steps can be walked through backwards. This insures that to go one step too far to observe something does not force to redo the experiment from start, like a cake that has been cooked too long; instead, it must be easy to step backwards. So doing, every user action is harmless, and can be undone. The user is free to test hypotheses and explore the program execution back and forth. As a consequence, steps and jumps can be executed forwards and backwards.

Finally, the user needs to make What-If experiments. For instance, when the teacher explains that the program must be such and such for some reasons, the learner is often not able to appreciate these reasons. In this case, it is better if the user can easily try faulty solutions, and observe what happens. This will help him integrate the teacher reasons instead of accepting them as God's words. So, it must be easy to edit a program and execute it.

2.2.2 Implementation

The source program is compiled in an intermediate language where IF, FOR and WHILE constructs are programmed using goto-like instructions. In this process, syntactic markers like `then`, `else` or `fi` become proper instructions. This simplifies the link between step actions and displays, in forwards and backwards directions.

The step-by-step behaviour is implemented by a stepper that takes a memory contents, an instruction number (the value of the program counter) and a direction (forwards or backwards), executes the instruction with that number in the indicated direction, and returns a new memory contents and a new instruction number. Jumps are executed by iterating steps.

The tricky part is to implement backward execution. A naïve approach is to count forward steps. When a backwards step is in order, the value of the step counter is read, say n, and $n-1$ forward steps are executed from the start of the program. As long as the user can only step, n cannot be very large since it represents physical actions of the human user, say n clicks on a STEP button, and re-executing $n-1$ steps cannot be detected by the user since the machine is so much faster that the user. However, this is no longer the case when the user can jump. A jump may hide many steps, and its execution may take a time which the user can sense. In that case, the user will be rather surprised

that executing one step backwards takes as much time as executing thousands steps forwards.

To avoid that we have designed the intermediate instructions so that they have a forward and a backward semantics. Usually, the backward semantics uses information that has been set by the forward semantics. For instance, the backward assignment must restore the variable as it was before the forward assignment. For this purpose, the forward semantics stores the old value of the variable before writing a new value on it, and the backward semantics reads the old value in the store to rewrite it in the variable.

However, the real protocol is a little bit more tricky because commands in a loop body can be executed several times. So, the forward executions of an instruction stores a journal of the information required to repeatedly executing them backwards.

To summarize, every instruction in the intermediate program is equipped with its own journal in which forward semantics writes what is useful for backward semantics. The actual contents of these journals depends on the instruction. For instance, assignment instructions record the journal of old values of variables, but `fi` instructions record the journal of the values of the condition in the corresponding `if` in order to be able to execute the IF command backwards.

As a result, backward steps always cost only one step.

2.3 An interface

2.3.1 Principles

The interface of the Waouh system is designed to be as simple as possible. A home page gives access to an interpreter page where one program at a time can be executed.

The interpreter page shows three areas.

1. The program area displays the program to be executed. The program is pretty-printed so that there is only one simple command per line. Every line can be clicked on to set breakpoints. The current line, i.e. the next line to be executed must be highlighted. This acts as a program counter.

   In this design, all lines behave as instructions, even lines with a purely syntactic keyword, like "`else`". This is to insist on the sequence of operation made by control statements.

2. The memory area displays the state of the memory. Every variable is displayed in a separate line, and the last assigned variable is highlighted. At the end of the execution, the result variables are also highlighted.

3. The control areas displays buttons to control execution of the program.

   An EDIT button opens an editor page where the program is displayed with the same layout, but can be edited. Instead of an EDIT button, the editor page has a COMMIT button to go back to the interpreter page.

The editor page is a simple editable text box with nothing more, no fancy IDE operations.

2.3.2 Implementation

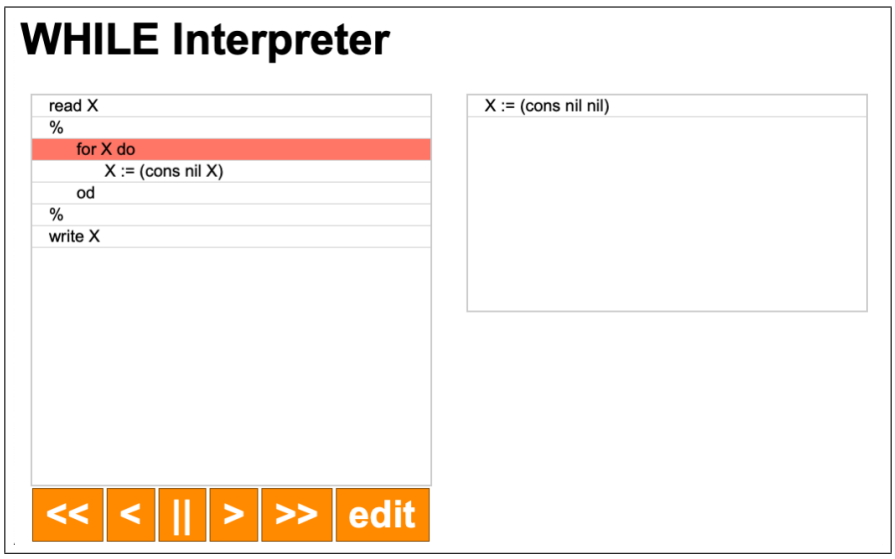

Figure 1: The interpreter page

Figure 1 shows the actual interpreter page. The program area on the left and the memory area on the right need no explanations.

The control area, bottom left, requires more explanations:

- The STEP button (>) permits to execute the current command and to go to the next line to execute.
- The BACK button (<) permits to undo the previous command and go to the previous line, in the backward execution order.
- The JUMP button (>>) permits to jump forwards to the next breakpoint. Breakpoints are set by clicking on a program line. If no breakpoint is set, the execution jumps to the end. If the jump enters an infinite loop, it will run forever, unless the user clicks on the PAUSE button (||).
- The JUMP-BACK button (<<) permits to jump backwards.
- The PAUSE button (||) permits to stop the execution process during a jump or a jump-back. It is required because WHILE loops open the way to infinite executions.
- The EDIT button (**edit**) permits to edit the current program.

When the READ statement is executed a popup box appears and the user can enter the input values in concrete WHILE syntax (Figure 2). A parser checks that the input is valid, and produces an internal representation of its denotation input (a value), and assigns it to the input variables.

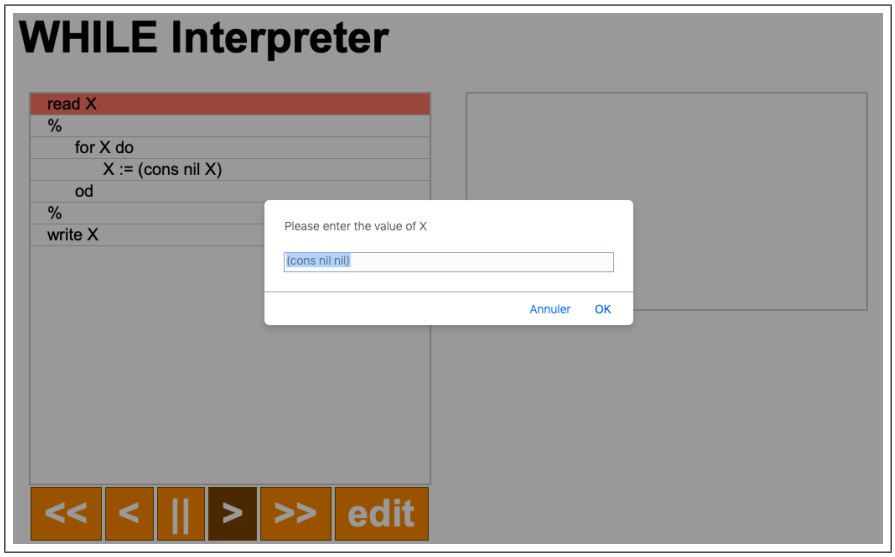

Figure 2: Entering an input value

Then, when the execution of the program is over, the output values are highlighted in green in the memory area (Figure 3).

If the user clicks on the EDIT button, the program area is transformed in an editable text box with the current program pretty-printed in it (Figure 4). The user can do all desired modifications in it. When the user clicks on the COMMIT button (**commit**), a parser checks the validity of the program syntax. If the syntax is OK, the intermediate representation is computed, then loaded in the program area, and the interface returns to the interpreter page. Otherwise, the interface remains on the editor page.

All the Waouh system has been written in Javascript and runs in a navigator. We are aware of portability issues between navigators, and we have tried to stay on the safe side of Javascript by using its strict mode, and disciplined object design as is possible since ECMAScript 2015 (ES6). We have tested the Waouh system with present versions of Chrome, Mozilla and Safari, but certainly more work is required to make the Waouh system really robust.

For writing the parser, we have designed a package that offers parsing tools in the style of Scala's parsing combinators [3]. This makes it easy to write a parser nearly as an algebraic grammar. "Nearly" comes from adjustements required to produce meaningful abstract syntax trees and meaningful parsing errors.

The grammar/parser is quite easy to write, thanks in particular to the very

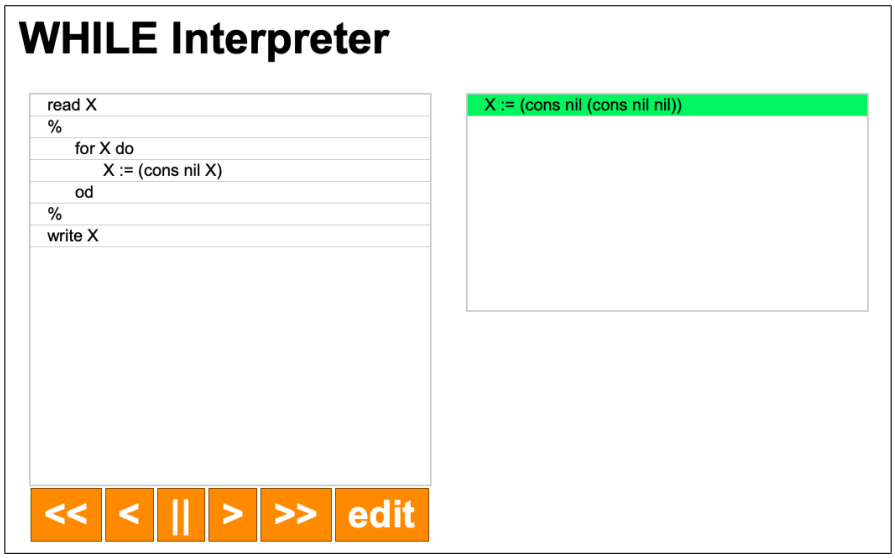

Figure 3: Highlighting an output value

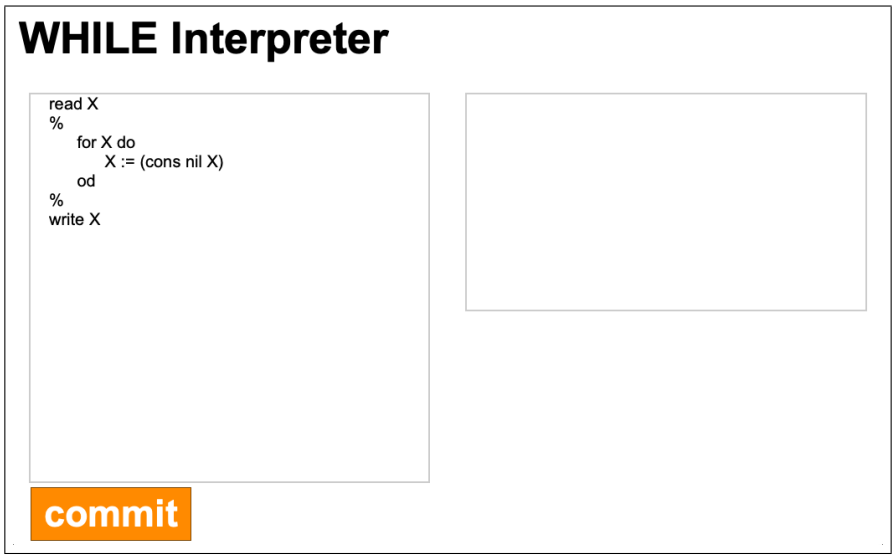

Figure 4: The program editor

explicit syntax with a lot of ponctuations like `read`, `%`, `write`, or `if`, `fi`, `do`, `od`. This also helps in producing meaningful parsing errors.

2.3.3 A short visit

Let us take the example of figure 1. In essence, this program adds its input to itself; it doubles its input.

1. The user steps once to execute the read statement. The answer is an invitation to write a value for the input variable (Figure 2).
2. The user enters `(cons nil nil)` (which can be interpreted as 1) and clicks the OK button.
3. The user steps until the program is finished. Stepping too much will do no harm. The output is highlighted in the memory area (Figure 3). The result is `(cons nil (cons nil nil))` which can be interpreted as 2, as expected.
4. The user clicks on the first line (`read X`) to set a breakpoint.
5. The user clicks on the JUMP-BACK button (`<<`). Execution starts in reverse and stops at first line.

3 Conclusion and further works

In its current state, the Waouh system is designed to experiment with the very beginning of programming. As a consequence, it offers nothing for learning more refined concepts. It is enough to play with the concepts of the computability theory course of the university bachelor program, but is not enough for other courses that present more advanced programming concepts.

Therefore, we plan to extend the Waouh system as a stack of increasing complexity levels. For instance,

1. WHILE 0: basic WHILE programming language;
2. WHILE 1: WHILE 0 plus functions considered as encapsulated WHILE 0 programs;
3. WHILE 2: WHILE 1 plus modules considered as static encapsulated WHILE 1 programs;
4. WHILE 3: WHILE 2 plus objects considered as dynamic WHILE 2 programs;
5. etc.;

The challenge will be to insure that the design of the Waouh system is modular enough so that the stack of concepts is also a stack of programs that implement it.

Another extension is to integrate a blog-like functionality so that a teacher can post an explanation and a program, and the learner could read the explanation and execute the program. Conversely, learners could post programs and their explanations, to be checked by the teacher. In the same didactic direction, a teacher could create challenges that learners could try to solve. This permits group learning, and introduces an emulation.

In order to emphasize the need for program validation, learner submissions could be either programs or test cases. A test case that kills an incorrect program would be considered as an interesting answer. Going a little bit further, one could extend the language with assertions to encourage the expression of formal properties. A program that fails to pass an assertion would be considered better than a program that fails to pass a test.

In the same vein, simple complexity experiments could be realized by recording all executions of the same program by different users. Learners could be challenged to identify best cases and worst cases: my best case is better than yours!

As it is based on the same WHILE variant as [1], the Waouh system could also be adapted to explore theoretical issues. For instance, many WHILE examples given in Neil Jones's book could be executed for real, and in real size. Similarly, [4] explores different equivalences between WHILE programs, Turing machines, and other programming models, that could be animated in the Waouh system.

References

[1] Neil D. Jones. *Computability and complexity - from a programming perspective.* Foundations of computing series. MIT Press, 1997.

[2] Albert R. Meyer and Dennis Ritchie. The complexity of loop programs. In *ACM 22nd National Conf.*, 1967.

[3] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12.* Artima Incorporation, 2016.

[4] Favio Ezequiel Miranda Perea, Araceli Liliana Reyes Cabello, Rafael Reyes Sánchez, and Lourdes del Carmen Gonsález Huesca. La máquina de Turing en el ámbito de los lenguajes de programación. *Miscelánea Matemática*, 56, 2013.

[5] John T. Stasko, Albert N. Badre, and Clayton Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In Bert Arnold, Gerrit C. van der Veer, and Ted N. White, editors, *INTERACT '93, IFIP Int. Conf. on Human-Computer Interaction, jointly organised with ACM Conf. on Human Aspects in Computing Systems CHI'93*, 1993.