GrandeOmega
A didactic model for learning programming

Francesco Di Giacomo, Giuseppe Maggiore
Mohamed Abbadi, Marijn Bom

Abstract

Teaching (and learning) how to program is an open problem. Both teachers and students cite a variety of issues, ranging from difficulty bridging the gap between theory and practice, to lack of simple assignments suitable for beginners. We believe that technology can offer a powerful tool in supporting a solution to these problems, but at a high cost in internal complexity. For this reason we have built GrandeOmega: a generic, web-based code-interpreter that can be easily used to build interactive lectures and assignments where students experiment with learning how to program while receiving real-time feedback.

1 Introduction

Learning how to program is a daunting task. There is a myriad of studies [...] documenting all sorts of didactic approaches in teaching programming in both lower and higher education by means of games, flipping the classroom, role playing, massive online courses, and much more.

Unfortunately, a clear winner has not yet emerged. The broad variety of methods is perhaps a symptom of a still open problem, which we believe is still very much present and made even more urgent by government calls to increase the number of graduates [...] to keep driving knowledge economies.

We observe that in many institutions, either the survival rates of students are quite low [...], or (in some extreme cases) the quality of the curriculum is adjusted downwards to avoid punishing students too much. Feedback from companies around the authors is often quite damning: the quality of graduate programmers is often too low, citing lack of understanding of basic aspects of programming such as memory models, concrete semantics of languages, types and type systems, with the result of slow, bug-ridden code being shipped all too often [...]. In some places we are even witnessing a slow fading of the engineering from software engineering.

We propose that learning to program is indeed fundamental, but it must happen at a high level. Those who will build the digital infrastructure of tomorrow must be able to do so with reliable, high-performance results: an infrastructure that barely works and holds together will be worthless.

Problem statement: the broader issue that we try to tackle in this paper is the study and improvement of teaching programming in higher education institutions.

To do so, we will begin by analysing the issue of how complex is it to learn how to program (Section 2). We will then provide a sketch of our solution with all the ingredients we believe to be fundamental for solving the issue (Section 3). We take the ingredients together and discuss the architecture of our actual implementation of such a solution (Section 4). Finally, we provide an evaluation of the impact of our (large) trial at the Rotterdam University of Applied Sciences (Section 5).

2 Why is it so hard?

Within the context of higher education, learning programming is hard for both beginners and students with past experience.

It is suggested by some [...] that learning programming is no different than most other complex skills: it takes roughly ten years (ten thousand hours) to become truly proficient.

The reason why it takes so long is disarmingly simple. Programming requires both the ability to *understand* and to *design* code.

2.1 Understanding code

Understanding code is a passive skill, but not any simpler because of it. The true meaning of code is the sequence of steps that the machine will actually perform: every single bit that will be read and written as a result of an instruction is part of the meaning of that instruction, just like every cache hit-or-miss, the activation of the CPU ALU(s), network channels, operating systems, interpreters, just-in-time compilers, and ultimately interactions with users. Being able to figure precisely what a program does, and how it does it (also in terms of performance) requires the ability to formulate an abstract idea of the program behaviour, and the mapping of this *abstract idea* to the concrete components when more specific reasoning is needed.

The sheer size of the machinery involved in the execution of even the simplest program is simply *very large*, and *the ability to think hierarchically and zoom in and out of the details as needed takes a lot of experience.*

2.2 Designing code

Designing code is an active skill, and as such intrinsically complex. Designing code requires the formulation (and therefore the choice) of a design strategy, usually in a top-down fashion, which is then recursively turned into a more and more concrete definition of the program. Being able to design a program effectively requires the ability to choose a specific design among a series of

possible designs, which taken together form *the abstract meta-strategies* that characterise a programmers knowledge, style, and experience.

The sheer size of the design space of even the simples program is *so large as to be essentially infinite* (we are talking about finite machines after all!), and the ability to *formulate meta-strategies and employ them recursively takes a lot of experience*.

2.3 Size matters (and so does structure)

We believe the size of the domain to be the core of the issue. Even though some students might already know a few tricks to produce working programs in some very narrow domain, the fundamental ability to abstractly reason about code (both for understanding and designing programs) is usually severely lacking in first year students.

Moreover, we cannot just solve the problem by throwing unstructured assignments such as read this code or try and write this program, as we must train the specific mental activities that we wish to stimulate in students. Specific training must be structured in order to gently guide the activation of the proper thought structures, in a slow buildup of complexity and freedom to express ones own creativity.

2.4 The issues

We close this session by identifying a series of practical, concrete issues that we believe sum up the discussion so far:

ID	Issue
REASON_MODEL	Students need extensive practice with reasoning about models of programs
REASON_DESIGN	Students need extensive practice with reasoning about existing design strategies
EXTEND_DESIGN	Students need extensive practice with extending existing programs (which should follow a formative design).
EXTEND_BUILDUP	Students need a buildup in complexity with their extension activities.

By keeping these issues in mind, we will now setup a proposed didactic model which we believe can mitigate them or outright resolve them.

3 The ingredients and the didactic model

The didactic model we propose (just like the implementing software) is called GrandeOmega.

Whereas many revolutionary didactic models completely remove emphasis from the role of the teacher, thereby forsaking the richest source of knowledge and explanation available in the vicinity of students, GrandeOmega sets out to empower both teacher and student.

The teacher sets up his course, in a way similar to traditional lectures. Each lecture is made up of slides, and these slides follow some visual formalism to explain code. One of the most used such formalism is (recursive) tables of names and values. Traditional slides are *passive*.

Some of the slides, ideally one every quarter to half an hour, are active slides. The active slides use the *very same formalism of the passive slides, but with a major difference: students can experiment with them.* This means that the theory discussed by (and with) the teacher is not separated from the practice, but comes to life and strengthens the practice just as much as the practice gives context and use to the theory.

Experimentation within active slides takes two forms: *forward* and *backward* assignments.

3.1 Forward assignments

Within forward assignments students practice their understanding of code. They get to see whole (small) programs, and they are asked by the system to provide the values of variables or constants. The values that need to be predicted by students can range from the integer value of a global variable (`x=1`), to the value of a reference in the virtual heap (`l=ref(10)`).

3.2 Backward assignments

Within backward assignments students practice with design strategies to build code. They get to see whole (small) programs where some bits have been hidden, and they also get to see all the steps that the complete program would take. Students must guess back the hidden bits of code. The hidden bits of code range from a constant (`5`) or a variable name (`x`) to whole expressions (`i*2`), to whole instructions, functions, methods, classes, queries, potentially up to the whole program.

3.3 Immediate feedback

Each and every mistake automatically produces immediate, visual feedback in the form of colors and icons. The input from students *is evaluated, not just compared as text.*

3.4 Gamification

Progress of a student, and every success, is stored and clearly shown in context by means of colors and icons. A student can see, at a glance, how far he is: whether lagging behind, or following nicely.

3.5 Insight to teachers

All the *data gathered* by the logging systems is *analysed and presented to teachers*. This makes it possible to identify difficult topics (assignments where the majority of students is getting stuck), identify struggling students (who are getting stuck in most of the assignments), and to identify negative behaviours (too few hours of study, lack of regularity, etc.). As stated above, we strongly believe that modern didactics should not try to provide a revolutionary removal of the teacher: *the teacher remains the director of the educational orchestra, even though lots of active work is performed by the students.*

3.6 Extra ingredients

Understanding and designing code on a smaller scale, with a gradual buildup in size and complexity, makes it possible for students to gain a deep understanding of the underlying logical mechanisms.

To further increase efficiency of a curriculum, the task of learning should be cornered from multiple sides. Students should also be presented with the challenge of freely experimenting with open designs and projects to build. Thus, the didactic method as a whole can be summarised as:

- forward assignments to learn *understanding*;
- backward assignments to learn *design*;
- projects to *sum it up.*

4 Technical details

The web-based implementation of GrandeOmega features, at its core, a generic code interpreter which takes as input a specification of one or more programming languages, and yields the following as output:

- an interactive, assistive code-editor (Figure 1);
- a visual debugger (Figure 1);
- a forward assignment editor (for teachers only - Figure 2);
- a backward assignment editor (for teachers only - Figure 2);
- a slide editor (for teachers only - Figure 3);
- a slide presenter (for teachers and students);
- a forward assignment environment (for students only - Figure);
- a backward assignment environment (for students only).

The system also features a dashboard which shows assignment data over a whole course, a whole class, or a single student.

4.1 Architecture

In order to ensure the right level of responsivity, the application is features a large front-end written in React and TypeScript. React is a UI frame-

Figure 1: Code editor and debugger

Figure 2: Teacher's assignment editor

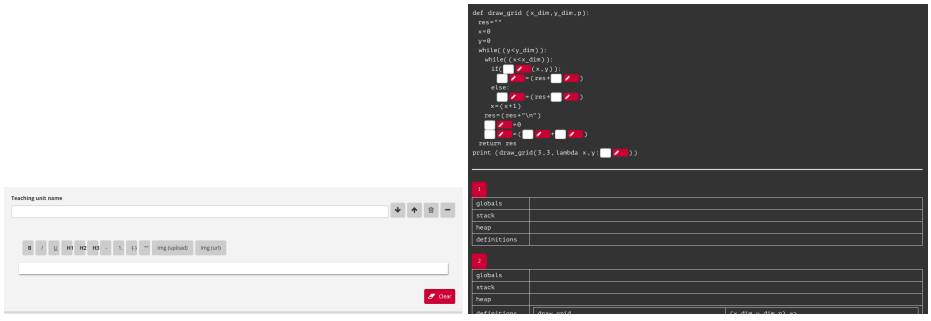

Figure 3: Slide editor and student's editor

work published by Facebook, implementing the functional reactive programming paradigm [...] in JavaScript and for browsers. The combination of referential transparency and controlled state updates make it possible to build truly complex, deeply hierarchical interactive structures with a high degree of correctness and confidence[add reference to referential transparency correctness]. TypeScript is an extension of JavaScript built by Microsoft, featuring a rich type-system inspired from the world of statically typed functional languages, emphasising inductive composition of types and generic programming. The combination of (functional) reactive programming and a rich language of statically checked types has enabled us to build a complex system with few defects and high performance and availability, without having to leave the platform of the web and therefore without sacrificing accessibility.

The backend is twofold: on one hand we have a traditional Ruby on Rails application which is quite simple in its data storage and retrieval operations. On the other hand, the same code interpreter that powers the front-end is invoked by the backend in the form of a Nodejs service (Nodejs is a server-side JavaScript execution environment). By means of Nodejs, both the frontend and the backend feature the very same code interpreter, avoiding very unpleasant mismatches such as a correct answer on the front end which maps to a wrong answer according to the back end, and viceversa.

4.2 The generic code interpreter

The code interpreter is the beating heart of GrandeOmega. First of all, there is no single programming language supported: GrandeOmega is extensible to support any programming language definable: from Python (currently implemented), to SQL, and further to mathematical languages such as the lambda calculus or even languages for set theory and boolean logic.

The code interpreter is based on pattern matching and substitution and it directly mimic the small-step semantics [add reference] used to describe the formal semantics of computer languages. The small-step semantics define a set of rules in the form of logical rules, which means they are made of a (optional) set of premises that, if satisfied, will produce the result defined in the conclusion. If a rule does not contain premises it is called *axiom* (meaning that its evaluation is immediate). Below you find an example of such rules describing the evaluation of the sum of two arithmetic expressions:

$$\frac{}{\langle \texttt{Integer x} \rangle \Rightarrow \texttt{Integer x}}$$

$$\frac{\begin{array}{c}\langle \texttt{left} \rangle \Rightarrow \texttt{Integer x} \\ \langle \texttt{right} \rangle \Rightarrow \texttt{Integer y}\end{array}}{\langle \texttt{left} + \texttt{right} \rangle \Rightarrow \texttt{Integer (x + y)}}$$

The first rule is an axiom: if the input of the rule is an integer number `x` then its evaluation simply returns itself. The second rule evaluates the sum of two

expressions by recursively calling the evaluation on the left and right expression. If the evaluation succeeds and returns an integer value, then the result of the whole rule is the arithmetic sum of the result of evaluating the two expressions.

From this example we proceed to define the general way of evaluating an semantics rule:

- The left part of the conclusion is analysed by using pattern matching in the following way:
  - If the current element is a variable then the pattern matching succeeds.
  - If the current element is a constant we compare it with the input of the rule and if they are the same then the pattern matching succeeds.
  - If the current element contains operators or other syntactical structures of the language (like the plus operator in the example), we compare the structure of the input of the rule with the one in the conclusion. This means that we check if the syntactical structure of the input of the rule is the same in the conclusion and then we check each argument of the syntactical structure by recursively applying these pattern matching steps.
- We try to evaluate each one of the premises in the following way:
  - We try to run each semantics rule in the language definition until one returns a result.
  - If the result of a premise (right part of the arrow) is a specific syntactical structure then we test the pattern matching.
  - If all the semantics rules fail to produce a suitable result then the current semantics rule fails to return a result.
- We generate the result of the current semantics rule.

4.3 The generic structured code editor

Moreover, the code interpreter is not only involved in the evaluation of programs, but also in the rendering of their debugging steps, and in the interactive, structured editor.

[[Mudy: add explanation here and a couple of screenshots]]

5 Evaluation

GrandeOmega has been tested extensively with students from Hogeschool Rotterdam, a university of applied science in the Netherlands. The classes were divided into two groups: in the first classes were given some programming assignments to be completed in the traditional way (without GrandeOmega), while in the second other classes were given the same assignments in GrandeOmega.

Class	Completions	Pass rate	Pass rate G.O.	Average grade	Average grade G.O.
INF1B	71.8	3	14	84.7	97.2
INF1F	56.7	6	5	77.0	88.1
INF1L	48.1	2	2	75	83.8
INF1A	41.7	7	7	82.1	86.5
INF1C	35.6	5	7	82.5	93.3

Table 1: Student performance before and after GrandeOmega

Class	**Correct prediction**	**False positive**	**False negative**	**Incorrect prediction**
INF1B	16	8	2	10
INF1F	11	2	4	6
INF1L	18	3	1	4
INF1A	13	0	2	2
INF1C	15	1	1	2

Table 2: Prediction of student performance

Table 1 contains the data relative to the pass rate and average grades of classes that used GrandeOmega before and after using the tool itself. Table 2 and Figure 4.

6 Conclusion

Class	**Grade**	**Passed students**	**Avarage grade**
INF1H	41.4	3	83.3
INF1E	30.6	1	75
INF1J	23.3	0	N.A.
INF1G	41.4	7	80.3

Table 3: Results of classes without GrandeOmega

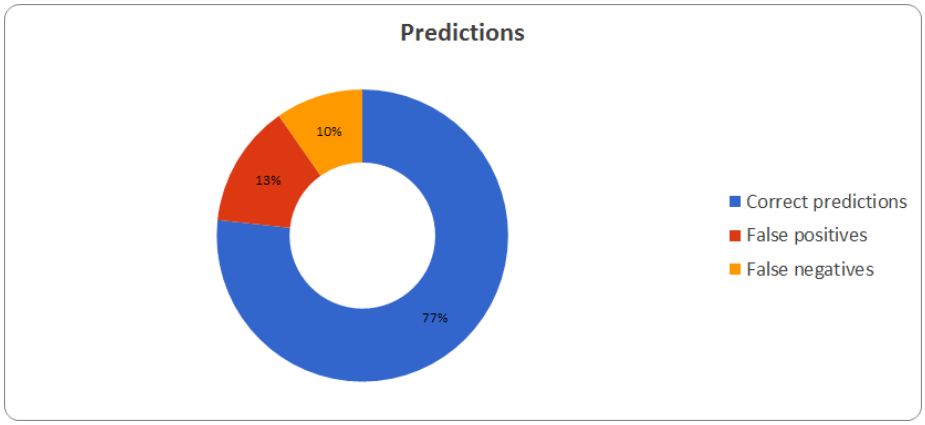

Figure 4: Predition accuracy of GrandeOmega