An Introductory Course on Programming based on Formal Specification and Program Calculation

Javier Blanco
Fa.M.A.F. - U. N.C.
Ciudad Universitaria 5000
Córdoba, Argentina
blanco@mate.uncor.edu

Leticia Losano
Fa.M.A.F. - U. N.C.
Cuidad Universitaria 5000
Córdoba, Argentina
al@hal.unc.edu.edu.ar

Nazareno Aguirre
F.C.E.F.Q.yN. - U. N. R. C.
Ruta Nac. 36 km 601 X5804BYA
Río Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

María Marta Novaira
F.C.E.F.Q.yN. - U.N.R.C.
Ruta Nac. 36 km 601 X5804BYA
Río Cuarto, Argentina
mnovaira@dc.exa.unrc.edu.ar

Sonia Permigiani
F.C.E.F.Q.yN. - U. N. R. C.
Ruta Nac. 36 km 601 X5804BYA
Río Cuarto, Argentina
spermigiani@dc.exa.unrc.edu.ar

Gastón Scilingo
F.C.E.F.Q.yN. - U. N. R. C.
Ruta Nac. 36 km 601 X5804BYA
Río Cuarto, Argentina
gaston@dc.exa.unrc.edu.ar

ABSTRACT

We report on our experience in teaching introductory courses on programming based on formal specification and program calculation, in two different Computer Science programmes. We favour the use of logic as a tool, the notion of program as a formal entity, as well as some issues associated with efficiency. We also review and use in practical cases some program transformation strategies, such as generalisation, tupling and modularisation.
We describe our approach, its advantages and drawbacks. Furthermore, we present some preliminary results from an ongoing qualitative research which intends to characterise, describe and understand the students' experiences when taking these courses.

Categories and Subject Descriptors
k.3.2 [**Computer and Education**] Computer and Information Science Education – *computer science education*.

General Terms: Algorithms

Keywords: Computer science education, Formal specification, Program derivation and verification, Qualitative research in education, Functional programming.

1. INTRODUCTION

It is generally agreed that teaching introductory courses on programming is a very difficult task. Often, such courses have various different aims in Computer Science curricula, besides providing students with the basics of programming. Some of these «extra» aims are training students in some of the necessary technologies they will need in later courses, and provide a glance of a bigger picture in software development, and the many challenges associated with it. It is not surprising then that many of the current approaches to a first course on programming are related to what is thought to be more useful to students in their later programming practices, typically strongly based on modern and sophisticated programming languages such as Java, and including small to medium size programming projects where students can experience, to some extent, some typical activities in software development (e.g., separated phases for analysis, design and implementation, the importance of modularity, testing, etc) [15, 16]. This situation generally leaves lecturers with limited time to teach students the complexities associated with "programming in the small", and concentrating on reasoning about small programs (it is not surprising then the growing belief amongst practitioners that dealing with programming in the small is *easy*). Also, it seems that a now popular widely spread approach to teaching programming in the small is associated with *structured programming*; moreover, sometimes structured programming even comes with an emphasis on program verification.
In these cases, however, program verification is usually taught as a task to be performed *after* one constructs a program, thus contributing to make students believe verification to be an additional "burden". Furthermore, due to the fact that one usually picks relatively simple problems for teaching programming and verification in introductory courses, this leaves students with the feeling that verification is not only additional burden, but also *optional* burden (students get the idea that they have to verify programs that they already know to be correct).
In this paper, we report on our experience in teaching introductory courses on programming based on formal specification and program calculation, in two different Computer Science programmes, at the National Universities of Córdoba and Río Cuarto, in Argentina. The motivation of these courses are overcoming some of the previously described problems, which were experienced in our programmes. Our current approach is characterised by the use of logic as a *necessary* tool, right from the start in

the formal specification of problems, and as part of the body of rules for transforming these specifications into programs. In order to make the transformation smooth, and not having to deal with the complexities of imperative languages, most of our courses are based on functional programming. The connection to imperative programming, although limited, is based on program transformation. As it will be described later on, we skip many of the important "programming in the large" issues, as well as technicalities associated with imperative (or object oriented languages). We favour instead the use of logic as a tool, the notion of program as a formal entity, as well as some issues associated with efficiency. We also review and use in practical cases some important program transformation strategies, such as generalisation, tupling and modularisation. Also, as it will be explained more throroughly later on, we try to carefully choose the exercises, trying to emphasise the cases in which problems would be extremely difficult to solve if they are not formally manipulated. We describe our approach via one of these cases, the segment of minimum sum problem in section 3.

In this context, we are carrying out a qualitative investigation whose primary objetive is characterising and understanding the learning processes associated with programming in the above mentioned introductory courses on programming, particularly that of the University of Córdoba, where the course is taught to first-year students. It is considered that in the interaction between students just starting the Computer Science programmes and the instructors two important learning processes are produced, namely learning important abilities such as modelling, abstraction and arguments for program correctness, as well as learning the appropriate practices associated with being a good programmer (and a successful Computer Science student). As a consequence, this process involves a renegotiation of language and student identities. We will present some preliminary results of this investigation, particularly associated with three problems, namely the *insertion of students into a new community*, *learning the programming languages and paradigms*, and *understanding and using mathematical proofs as tools for the development and demonstration of the correctness of computer programs*.

2. DESCRIPTION OF THE COURSE

As explained before, the course we are describing here is an introductory course on programming taught at two different Universities. In the National University of Córdoba, the course is taught during the first two semesters of a 5-year Computer Science programme. In the National University of Río Cuarto, on the other hand, the course is taught during the third and fourth semesters, again of a 5-year Computer Science programme. In the former, students have no previous courses on programming or logic, but they take simultaneously with this course one on discrete mathematics. In the latter case, when students start the course they had already taken a 2-semester introduction to imperative programming course, and a basic course on mathematical logic. Although the course is taught in different contexts in the two Universities, we seek achieving the same general goals.

This course is based on two different traditions: formal derivation of algorithms and functional programming. We follow the idea of teaching the first course in programming based on a functional programming approach. One of the early documented experiencies was done in the University of Twente (the material used in a later version of that course can be found in [9]). Functional programming is easier to reason with, since the absence of an implicit state allows the use of pure equational logic and syntactical substitution on the programs. Furthermore, proofs by induction are the main tool to both verify and formally calculate programs. The a posteriori proof and the formal derivation are close enough (actually they are almost the same) for the student to recognise that he or she is doing both programming and verification at the same time. The style of reasoning (e.g., see [8]) is coherent with the one that will be used later on in the course to develop imperative programs, showing that although the computational models are rather different, at the programming level some concepts and many abilities can be reused when switching paradigms. This part of the course is based on the long tradition of imperative program calculation started by E.W. Dijkstra (cf. [5, 10, 6, 13, 4, 1]) and is standard with the exception of some particular attention to tail-recursion as a linking tool between both paradigms.

We tried to keep the new concepts to a minimum and still be able to solve many programming problems, avoiding if possible ad hoc solutions and using basic mathematical principles -induction, syntactical substitution, equational reasoning, fold/unfold- instead of an operational approach. Since for some of the subjects treated during the course there was little teaching material available (in particular in Spanish) when we started the project, one of the authors of this paper and other instructors wrote a book [2] which focuses mainly on the new approaches, i.e. formal derivation of functional programs from specifications in Dijkstra-Feijen's style of logic and very little use of higher order functions, as well as the translation from functional programs into imperative ones.

Specifically, we want the students to accomplish the following goals:

- Develop the ability to formalise problems, using logic as a tool.
- View specifications and programs as formal entities, and consider programming as the manipulation of these formal entities.
- Obtain considerable training on using recursion as a powerful mechanism for defining functions/programs.

- Understand that reasoning about functions can be exploited not only in functional programming, but also in imperative programming (particularly via transformation schemata).
- Get acquainted with a very elementary theory of abstract data types (and its relevance in program development).

The contents of the course is composed of the following three main modules:

- Logic and specifications. We employ an equational version of predicate logic with generalised quantifiers [7] (see also [8, 6]). The main goal is to have a suitable tool for dealing with large formulae (mainly programs). This logic is used for both the functional and the imperative paradigms.
- Construction of functional programs. The students learn how to formally construct functional programs from specifications, with their corresponding inductive proof of correctness. We make an induction guided use of the *fold/unfold* rules, in order to guarantee termination in the calculated programs, as in [12]. Operational reasoning appears only as a motivation for the axioms of the calculus, and for efficiency considerations. In this paradigm, programs and specifications are written in the same formalism (programs are a subset of the possible formulae) [12].
- Construction of imperative programs. This module is rather traditional. We try, however, to use what was learned in the previous module to help in this process. The main tool is to translate functional programs into the imperative formalism by using *tail recursion*, which not only allows us to translate the program itself, but also its proof of correctness. Although the underlying computational models are different in the functional and imperative paradigms, the students can get the feeling that proofs in the two contexts share similar ideas (e.g., invariants can be seen as a restricted way to use induction). Also when introducing imperative programming, students are faced to the notion of abstract data type. We employ the case of lists (which are somehow inherent to functional programming) and an array based implementation of lists in imperative programming, to show how functional programs handling lists are transformed into imperative programs manipulating arrays. The usual formal concepts of abstraction function, representation invariant and the like are used superficially.

3. A SAMPLE EXERCISE

In order to better illustrate our approach, and the notation used, let us provide an example, which is an exercise used in the course. Consider the problem of, given a list xs of integers, finding the sum of the elements in the segment of minimum sum of xs. A segment of xs is simply any sublist of xs. So, for instance, if list xs is [1,-4,-2,1,-5,8,-7], then the minimum sum segment of xs is [-4,-2,1,-5], and its sum equals -10; if we consider the list [1,2], then its minimum sum segment is []. A first step is to formally specify the problem. For this task, the specification language we use provides generalised quantified expressions (with general rules for dealing with these), which can be built out of any binary operator, as long as it admits a neutral element, and is associative and commutative. This style is similar to that used in [8]. The generalised expression corresponds to applying the operator under consideration to a range of values. The operator Min satisfies the above conditions, and therefore can be used in a quantified expression, allowing us to straightforwardly specify the problem in the following way:

$$minSum.xs = \langle \text{Min as, bs, cs} : xs = as +\!\!+ bs +\!\!+ cs : sum.bs \rangle$$

where *sum* is a function that computes the sum of the elements of a list, for which we already have an operational version. Deriving a recursive function from the above specification is done via induction on the length of xs. A detailed calculation of function *minSum* can be found in [12], page 147. The resulting recursive function is the following:

$$\begin{array}{ll} minSum.[] & = 0 \\ minSum.(x \triangleright xs) & = g.(x \triangleright xs) \text{ min } minSum.xs \end{array}$$

where *g* is defined as follows:

$$\begin{array}{ll} g.[] & = 0 \\ g.(x \triangleright xs) & = 0 \text{ min } (x + g.xs) \end{array}$$

From these functions, we can do various things. For instance, we can employ schemata for transforming the above functions to tail recursive versions, and from the resulting functions straightforwardly obtain imperative programs. Also, we could attempt to derive a more efficient version of *minSum*, using transformation strategies (in this case, tupling is a suitable one). The resulting more efficient version of *minSum*, obtainable using tupling on *minSum* and *g*, is the following:

$$\begin{array}{ll} h.[] & = (0, 0) \\ h.(x \triangleright xs) & = ((x + b) \text{ min } a,\ 0 \text{ min } (x+b) \\ & \qquad [\![(a,b) = h.xs]\!] \end{array}$$

4. RESEARCH METHODOLOGY

The employed research methodology is of a qualitative nature, i.e., it is centred in a deep comprehension of a social phenomenon and not in its measurement. We believe that this methodology is well suited, since it will enable us to examine the various perspectives of a situation centred around experiences and personal processes, such as

learning, comprehending, teaching, deciding, etc., which are naturally descriptive.
The descriptions of these phenomena are being developed based on the analysis of the written material produced by students, protocols of personal interviews, and observations during course lectures and tutorials, taken during 18 months of field work. The gathered data is being analysed via an inductive/constructive process [14]; from this data and via this process, categories and conjectures are proposed, whose validity is later on tested. This process enables us to exploit the gathered information to elicit a grounded theory whose categories should be *"applicable -not forced- and indicated by the data in the study (...) [and] significantly relevant for, and able to explain the behaviour of, the study"* [14].
It is important to remark that these methods aim at allowing the researcher to interpret the gathered data from the perspective of those involved in the investigated situation, i.e., to understand the meaning that the various participants associate with the phenomenon under consideration. Thus, we believe that this methodology allows us to present, in detail, the opinions and views of the participants, expressed in their corresponding produced material [11].

5. PRELIMINARY ANALYSIS OF THE DATA

We carry out this preliminary analysis based on observations during course lectures and tutorials, conversations with instructors, talks given by former students, and some advanced students, reviews of mid-term and final exams, and interviews with three students who finished the course, with a good performance. As we mentioned, we identified three main relevant problems: mathematical proof of program correctness, programming languages and paradigms, and insertion of students into a new community. The relationships between these problems is an issue to be resolved in later analyses. We consider these problems from the points of view of three involved groups of people: instructors, students, and former students who took the course a few years ago, and who have already graduated.
Next, we will concentrate on the first two of the above mentioned problems. A detailed analysis of the third one is part of our work in progress, and we will describe it only superficially in this paper. In the quotes below, we have marked those corresponding to students by ST, whereas quotes marked as GR correspond to graduates.

5.1 Point of view of the instructors

Let us first describe our point of view, i.e., that of the instructors. We have been using the described approach for the past 10 years in the National University of Córdoba, and for the past 6 years in the National University of Río Cuarto. The experience we gained along these years enabled us to improve the course in various respects (particularly, collect better exercises, with interesting non trivial solutions, illustrating some of the benefits of program calculation), and observe various advantages and drawbacks associated with the course.
As advantages, we can say that students who get to assimilate the principles taught in the course have demonstrated to incorporate these in their programming practices, in particular in later courses. Also, although they generally do not use formal approaches in later programming courses, the acquired skills in logic and program manipulation leads to producing better programs (with fewer bugs and clearer), and to a more careful reasoning when programming. One important drawback related to demostrations is that the students usually becomes too "syntactic" in reasoning about programs, which has a negative impact in abstraction. It is usually rather difficult for students to "jump" in and out from the calculus (i.e., take perspective on the situation of the derivation at hand, and decide accordingly).
With respect to entering a new community, many lecturers consider that students have difficulties in founding their studying activities based on problem solving, as opposed to the more traditional approach organised around learning definitions and theoretical concepts; instructors generally feel that it is very difficult for students to anticipate how much time they will need to dedicate to studying. As a consequence, instructors feel that students cannot achieve the rythm of study required in University, in this kind of courses.

5.2 Point of view of the Graduates

Three graduates from Fa.M.A.F. gave talks to first year students of the National University of Córdoba, from which we could gather, at least partially, their points of view regarding the skills acquired in the degree, and the status that graduates from Fa.M.A.F. have gained in the local software industry.

> *"I am pleased with my career, I think that this is a good place where to learn a lot of things, which go beyond learning a particular programming language or tool suited for a particular task. What is good is the framework (...) all these things that you are being taught right now and ask yourselves to what purpose (...) we are using all these (...) When you understand how to think about these things, it becomes much easier to solve more complex problems. Now you are learning this, and you just can't imagine how complex it becomes as time goes on" (GR1).*

> *"Something really good about studying here at Fa.M.A.F. is that you receive a technical training, a*

training on the basis, that will allow you not to be tied to the technology currently popular (...) Here you receive a training on the basis that will allow you to do something that is essential in the software industry, namely the ability to be constantly learning new technologies" (GR2).

From these opinions, we can say that the graduates consider that learning programming paradigms and formal manipulation are two closely related kernels of study. The associated ability to apply methods and mathematical concepts in program construction is what later on allows them to distinguish as professionals. It is difficult for students to understand the relevance of these topics of study during the first year of the programme. In fact, these talks given by graduates aim at trying to facilitate the insertion of students in this new community; this is important, in particular due to the heterogeneity of the vocational expectations of first year students.

5.3 Point of view of the students

With respect to the category *insertion into a new community*, it is rather surprising that students consider that the previous education, that obtained during secondary school, is not very important. It seems that the nature of the contents, and the methods of study, in secondary school are disconnected, or at least differ substantially, from those used in the first year of study in the University. This would suggest a strong discontinuity between the contents, methods and justifications of the University and secondary school. Students also have difficulties, particularly during the first months in University, in maintaining the required rythm of study. With respect to this issue, students consider that every new subject of study requires a great learning effort, which generally comes right after the previous subject has been comprehended, and thus leaving a sensation of having no time for relaxing. Also, students remark the importance of meeting with students in their second and third years of study, as a way of identifying themselves with people who have been through first year already. These meetings, held in the form of talks, would allow first year students to realise the changes experienced by people that belong to the University.

Even though students get to develop complex computer programs, including implementations of datatypes such as dictionaries, addition and multiplication of polinomials, etc, they generally think that their programs are "toy implementations", thus feeling a separation between the "real world" of programming, and that learned in the University. One of the interviewed students, who had a previous experience in imperative programming, seemed to have more learning difficulties than the rest, perhaps due to his previous knowledge regarding programming being structured around an operational model, and its associated way of thinking about programs. This issue is consistent with what is observed in the case of students from Río Cuarto, where the course is taught in the second year, and where surprisingly the results have generally been slightly worse than in Córdoba. We believe this might be due to the case that, for most students, the approach to programming taught in the course feels somehow contradictory to the way they are use to do programming. They also exhibit a greater resistance, compared to students in Córdoba, to learning and applying logic for specification.

With respect to the category *programming languages and paradigms*, the students found particularly difficult the move from functional programming to imperative programming. They related these problems both with the specific aspects of a new programming language and the model of abstract machine of a new paradigm. Similar difficulties are also observed in [3].

"In functional programming, our programs were always pretty nice stuff, a type declaration, a small function and that's it; in imperative programming we had a huge bunch of things, constant declarations, variable declarations, semi-colons, printf's, etc. Then we started learning that semi-colons are necessary because we need to indicate the computer where things are separated, printf's are useful for printing stuff on the screen, variables are needed because the computer needs to know which things are going to be variable" (ST2).

"Up to this point, in functional programming, we had seen the computer as function reduction, now we saw it as state change. Change of what? Of state. But what is a state? Such and such. And already in the next lecture everybody was talking about state, and I had no idea about it (...) that variables are modified, and that the state is the value of the variables at a given moment in time" (ST2).

Finally, students also remark that courses are rarely oriented to programming languages, and that these do not cover the details of the programming languages used. This makes them feel disoriented, and in many ocassions cause them difficulties in progressing with their work due to a lack of knowledge of a set of commands, pre-defined functions/routines, and other available language constructs:

"When we were programming, and a new set of exercises was available, there usually was a lot of tools that we needed related to the language, and that were not taught. While we were trying to solve the exercises, the instructors gave us some tips, but we still had that feeling of being lost (...) For example, the issue of C libraries, compiling, etc (...) what makes it hard is that you need new things in the language which are not taught (...) You start

programming and you get to a point where you ask yourself 'how do I do this?' You ask an instructor and he says Ah! You need to use such and such functions from the Prelude, or load a particular library (...) During the lectures we see things more in detail, but when you have to actually implement things is not the same, even if you know what a loop is, if you don't know the commands you need to use and their right syntax you can't do it (...) you always encounter this kind of minor details, which are not minor at all" (ST1).

With respect to the problem *mathematical proof of program correctness*, we can make a few remarks. First, learning to handle the logical formalism that is used in the courses, and studying particular proof strategies, and using these for solving problems, enable the students, according to their opinion, to better understand the notion of mathematical proof in general, even as seen and studied in other courses. This became evident during the interviews, when we asked the students if they could draw some relationship between the courses related to programming and the courses on mathematics:

"There is some relationship (...) what was really useful for me was the courses on algorithms for other courses, particularly the topics on logic and propositional calculus. Because when they said 'it implies' or 'and', in the courses on algorithms I learned how to deal with all this, so when I was told 'this thing and this thing imply something else' (...) 'if this happens then this holds'. Well, that really helped, I mean what we learned in the courses on algorithms to grasp the essence of theorems, what they really mean" (ST2).

Having available a first-order calculus enables the students to deal with proofs at a syntactic level that is not exploited in courses related to mathematics in their programme of study, where proofs are dealt with in a semantic level. Students tend to absorve this difference as a concrete-abstract contrast:

"Using symbols (...) in mathematics (..) for us symbols are simply a formalism to express how you start, but we actually never, or rarely operate with symbols. In the courses on algorithms on the other hand, we do operate with symbols, 'it exists' means that there is some for which it holds, and that can be true or false. 'For all' means that it holds or it doesn't for all elements, and that can be true or false. In mathematics it is like this: 'let's see, let us suppose that if this holds, then this should also hold.' In the algorithms courses things are much more concrete. That relationship was very difficult to understand in the beginning, but once I understood it, it allowed me to better understand mathematics. [This would be related to] *the level of abstraction in which we deal with things. It is much less abstract in the algorithms courses, I mean the way we deail with mathematical symbols, don't you think? (...) We do not have summations to infinity, we have summations with precise bounds, or for the elements in a list, don't you think?"(ST1).*

Furthermore, students find it difficult to achieve a "creative" usage of the formalism employed in the courses. This creativity, we believe, would be related to the possibility of the students themselves being able to develop mechanisms for solving problems. According to this perspective, students who cannot develop these mechanisms, are limited to imitate those mechanisms presented by the instructor:

"One of the major difficulties that I had experienced was due to the lack of creativity that I had for tackling things (...) I mean, in discovering a method, in having original ideas, right? Because one can learn a system, a mechanical way of doing something, and you do it. But when you are told ' OK, let's see how you do this' and you have to create something, and in this creative process originality plays a great role, right? Of all the many possibilities of approaching the problem you have to choose one. I believe I'm quite limited in this respect" (ST1).

"Immediately after I start, in the first few topics, I feel a bit lost (...) And that's why I see as important the originality issue. In the beginning, even though I am given (...) the tools, I don't know how to use them, you have to be ingenuous to know how to use them, and I see sometimes that my friends in the class can do it (...) some very naturally, for whom the start seems to be much more smooth than for me. However, for those for whom the work is not that smooth, the way forward is seeing many times how it has to be done, so that it can be mechanised. The process of (...) grabbing the tools and ordering them for their use. That's the hardest part for me. In particular when during the course you are required to constantly change that process and the way you use the tools. That's when creativity comes into play (...) [While I progress with the sets of exercises] *that starts disappearing, and the difficulties only remain in very specific topics" (ST1).*

In order to adequately interpret the quantitative data associated with the success rate in passing these courses, it is important to take into account that public Universities in Argentina are free, generally with no requirements for registering for a degree. This is the case of all degrees

offered by Fa.M.A.F and U.N.R.C.. In these contexts, the desertion during the first year of study is typically high, independently of the degree in which the students are registered. Moreover, students can finish each course in their programme of study in two different conditions, as a regular (course passed) or not regular (course failed) student. They have then to pass a final exam, for which they have a number of available dates, along two years. Students that finished the course as not regular usually have to solve a few extra exercises, compared to regular students, but are still allowed to take the final exam. This situation makes it difficult to gather useful quantitative data, in order to compare our approach with other alternative introductory courses on programming. With respect to the performance of students during 2008, 191 students registered to take the course, 39 of which abandoned the course before the first examinations. Of the remaining 150, 43% finished the course successfully, i.e., as regular students. During the first three opportunities for taking the final exam, 59 students took it, and 37 passed it.

5. CONCLUSIONS

We have described an approach to a first course on programming strongly based on formal specification and program calculation, that we have been using in the National Universities of Río Cuarto and Córdoba in Argentina. Overall, we have been satisfied with the results. One of the main advantages we found is the use of very few conceptual tools (syntactical substitution, induction, equational reasoning, formalization and abstraction) which appear recurrently through the whole course and are comprehensive enough to develop a whole theory an practice of programming. Furthermore, these tools will be needed in many more courses in the curriculum, allowing the students to improve their understanding and mastery. The different techniques introduced prove to be useful to solve problems in unexpected ways (many programs obtained by these methods can only be understood through the construction steps). Moreover, the belief in the correctness of the programs strongly relies on the calculations and cannot be attained via operational reasoning. An early exposition to the usefulness of formal methods appears to be a cornerstone for developing the problem solving skills that are then acknowledged and required by the software industry.

We have found, via a qualitative analysis, three main problems. As future work, we plan to refine the analyses presented here, further exploring the relationships between the discussed categories, and enriching the work with new categories, that we believe will emerge as a result of the analysis. Another issue that we plan to explore is how students conceive the relationship between proof and derivation. This is considered crucial by the instructors of the studied courses, and could not be explicitly visualised by any of the students during the interviews. Of course, we hope that as a result of the analyses we will be able to reformulate and improve various points of the course. Furthermore, we also consider important integrating the contents and methods of the first two years of the programmes of study of Córdoba and Río Cuarto.

In the next few years, we are confident that we will have a more detailed and complete description of the phenomena described in this work. This would enable us to explain the differences between the conceptualisations of the various actors involved in the negotiation processes, regarding the significance of the discussed phenomena, as produced in the classrooms.

7. REFERENCES

[1] Backhouse, R. *Program Construction: Calculating Implementations from Specifications,* John Wiley & Sons, 2003.

[2] Blanco, J., Smith, S. and Barsotti, D. *Cálculo de Programas*, Fa.M.A.F, U.N.C., 2008.

[3] Clack, C. and Myers, C. *The Dys-Functional Student* in *LNCS* 1022, 289-309, Springer, 1995.

[4] Cohen, E. *Programming in the 1990s: An Introduction to the Calculation of Programs,* Springer-Verlag, 1990.

[5] Dijkstra, E. *A Discipline of Programming*, Prentice Hall, 1976.

[6] Dijkstra, E. and Feijen, W. *A Method of Programming*, Addison-Wesley, 1988.

[7] Dijkstra, E. and Scholten, C. *Predicate Calculus and Program Semantics*, Monographs in Computer Science, Springer-Verlag, 1990

[8] Dijkstra, E. and Schneider, F. *A Logical Approach to Discrete Math*, Monographs in Computer Science, Springer-Verlag, 1993.

[9] Fokkinga, M., *Werkcollege Functioneel Programmeren*, University of Twente, 1996.

[10] Gries, D. *The Science of Programming*, Monographs in Computer Science, Springer-Verlag, 1981.

[11] Hazzan, O., Dubinsky, Y., Eidelman, L., Sakhnini, V. and Teif, M. *Qualitative Research in Computer Science Education*. *SIGCSE Bulletin*, Vol. 38, 408-412, 2006.

[12] Hoogerwoord, R. *The Design of Functional Programs: A Calculational Approach*, PhD Thesis, Eindhoven University of Technology, The Netherlands, 1989.

[13] Kaldewaij, A. *Programming: The Derivation of Algorithms*, Prentice Hall, 1990.

[14] Lincoln, Y. and Guba, E. *Naturalistic Inquiry*, SAGE Publication, 1985.

[15] Liskov, B. and Guttag, J. *Program Development in Java: Abstraction, Specification and Objetc-oriented Design*, Pub-AW, 2000.

[16] Meyer, B. *Object-Oriented Software Construction*, Prentice Hall, 2000.