Course on a Mathematical Presentation of Functional Programming

Álvaro Tasistro Juan Michelini Nora Szasz
Universidad ORT Uruguay
{szasz,tasistro}@ort.edu.uy, juan@juan.com.uy

We present the motivations, syllabus and teaching method of *Foundations of Computing*, a course of Mathematics whose objects happen to be given by a functional programming language.

1 Motivation

We teach within a Software Engineering program which is 10 semesters long, with each semester comprising 16 weeks of lessons including examinations. We are in charge of the design and teaching of the courses of the Theoretical Computing Science area. The first course belonging to this area that students used to encounter was, until the beginning of the school year of 2013, *(Mathematical) Logic*, placed in the Second Semester of the program. It comprised the classical Propositional and First-Order material plus an initial part on Induction and Recursion. This part was necessary for the students to learn to define sets (notably, languages) inductively, and to compose recursive functions and proofs by induction on such sets, all of them methods to be used in the rest of the course and not taught elsewhere. Besides, ours was the only serious introduction to such matters prior to the Algorithms and Data Structures courses, which also belong to our area. As a consequence, the Logic course was divided into three parts of almost the same size, all of them a little bit too much compressed for the students to be able to fully incorporate the relevant material. In particular, we had become unsatisfied with the degree of understanding and mastery that the students were able to reach of the methods of recursive definitions of functions and their close connection to proof by induction. Reflection on the matter, together with the emergence of a timely reformulation of the entire program of studies, led us to realize that we were actually in the presence of an opportunity to address a combination of important issues, which we would like now to expose in detail.

The prime observation is that Induction and Recursion is of course a mathematical subject but, as such, it is possibly the one lying closest to Programming, so much so that it can be thought of being introduced as *a mathematical presentation of Programming*. This is made possible by letting programs be functions, generally recursive and acting on inductive data types, i.e. by considering some form of functional programming. Besides, we would treat these functional programs as the relevant mathematical objects, which means that their properties were to be investigated and proven, generally by induction. Going this way, we could then just take the path of promoting our Induction and Recursion material into a (first) course on Functional Programming, one that included the appropriate techniques of program proof. Sample courses and textbooks fairly meeting such description abound, so that setting up the syllabus and plan ought not to be a real problem. On the other hand, we would need a full semester slot to actually teach it, but that was fortunately put at our disposal, as a new course was created in the Second Semester, displacing our previous *Logic* to the Third. It is important here to mention that the new program of studies includes initial courses on Programming in the first two semesters, in which the Java language is

Submitted to:
TFPIE 2014

taught. As to the teaching of Programming, our new course was considered to come up as a convenient "second line", to be continued in subsequent courses in a way that we shall describe later.

Now, as it happens, additional considerations have led us to prefer accentuating further the mathematical character intended for our presentation of Programming. Specifically, we consider it most appropriate to expose the material in a rather strict mathematical style, by which we mean essentially following the pattern:

1. *Motivating considerations*, leading to the
2. *relevant definitions and constructions* and from those to
3. the interesting *theorems*,

that is characteristic of traditional Mathematics textbooks. The reasons sustaining this preference will be given presently, as we believe it to be one of the features specific to our approach. Before that, however, it seems convenient to comment on how such style fits within our actual teaching, since it looks at first rather harsh and unattractive, which might gravely affect its effectiveness. The ideal method of teaching that we have in mind is as follows:

1. Extensive course notes are produced, which follow the style depicted above. The notes include a good number of exercises, of varying difficulty.
2. The lessons assume the students to have read the corresponding material as previously indicated and are dedicated to the discussion and solution of selected problems. For doing this, the students organize themselves spontaneously forming working groups.

The number of students expected for this course is about 120, and it is divided into classes of about 30 students each. The first time the course took place was in the second (i.e. southern Spring) semester of 2013. On that occasion, we were actually not able to implement the preceding method entirely, due to reasons that, as we shall later explain, have to do precisely with the fact that it was the first time we were teaching the course. We are confident that this situation can and will be improved in future instances.

The reasons why we prefer to articulate the material in a rather strict mathematical style concern the kinds of approach that we consider appropriate for the students to experience to both Programming and Mathematics, which we turn now to explain.

1.1 The approach to Programming

As already said, we are talking about Software Engineering students. We believe that the practice of Software *Engineering* ought to include the achievement of *certainty* about the most relevant properties of every possible behavior of the program or system being designed or constructed. This requires thorough *understanding* of the material produced at each stage, be it specifications, models or code. Now, the *process* of getting to understand that the widget in question possesses a determinate feature can rightly be called a *proof* of such fact.

The difficulty with this stand is that *proof* is normally assumed rather strictly to mean a *text* written in a certain *form*. Indeed, proofs are most usually found in mathematical texts or lectures, and the manners in which they are redacted normally feature certain recognizable traits that warrant them as well-presented. The particular language may vary, depending on the discipline or style of the author. It could even be a fully formal one, i.e. a kind of code, in the cases where languages and formal systems of Logic are employed. Given this situation, we consider it necessary to stress the knowledge-enabling meaning

of proof, i.e. the *real content* that makes it a process or act of understanding[1]. It is in *this* sense that *proof is an indispensable part of Programming*. As to its written form, it is quite advisable to produce it in many cases for, if the writing is clear enough, it serves as a vehicle of understanding for others, including of course the very same author on a different occasion. Also it is the analogue of the calculations carried out in other branches of Engineering to ensure appropriate response of the construction performed and as such it ought, strictly speaking, always to be produced on demand. Sometimes, however, writing down large, uninteresting proofs might not be cost-effective or, when a formal system with an appropriate tool is being used, it can be downgraded to a just (almost) automatic process.

Now, when we turn to considering the education of first-year's Software Engineering students, we are led to stress the practice of proof in every sense of the word. That is to say, we must educate them in the *importance of understanding* what they are producing, as well as in *orderly methods for so doing* and, as a consequence of the latter, in *orderly manners of communicating or leaving trace of* the understanding achieved. One way of fulfilling these ends is to approach Programming systematically as a mathematical subject, so that programs arise generally as the objects of theorems. Mathematics consists precisely in presenting the relevant concepts in their full generality, and in deriving thereof the appropriate methods for carrying out further constructions and the necessary proofs. Of course it will be important to make the student aware and in command of the workings of mathematical practice, which actually constitutes itself in a requirement on how to teach such practice. This will be examined in the next paragraph on the approach to Mathematics.

Such systematic approach to Programming is certainly quite different from the one that, in our experience, the students pursue in their initial courses or can find in the texts to which they have access most directly. These consist too often of successions of *examples*, which are, for the worse, inevitably plagued with terms not formerly defined. As a consequence, the students receive the impression that there is nothing orderly about Programming, still much less that it could be possible to systematically *reason* about programs. They instead learn only to recognize certain code patterns whose meaning is never fully spelled out, and try to use them to produce whatever is required on a trial-and-error basis. They fail to transcend the particular cases, which affects their ability to form abstractions. And, of course, they are led to believe that testing their code in some more or less obvious particular cases is all that is required to ascertain its correctness. They do, anyhow, succeed in adapting the recognizable patterns for solving more or less accurately some problems belonging to a well-defined, bounded, catalogue. This is rather easily accomplished, which leads them to believe that there should be nothing really complicated as to Programming.

Students that participate of such thought are usually backed up by people ready to argue that the mathematical approach is cost ineffective, or directly totally unnecessary in practice. We, on the other hand, believe that reasoning about programs i.e. the approach to programming as a mathematical activity, is a matter of high educational value for:

1. The students have to learn that a well founded practice of Software Engineering is *possible*, after which they will expectedly discover that it is also worth pursuing.

2. It is at the basis of the employment of Formal Methods of software construction, which are inescapable in a number of applications and of increasing importance generally. They are at least undoubtedly worth being aware of.

3. It provides *discipline* of program development, which can be spelled out and is certain to improve

[1] Which could happen as one is convinced by another person's discourse, be it written or spoken, or as a result on one's own thinking and scribbling.

the quality of the students' production, at the same time that it actually helps to effectively solve many relevant kinds of problems.

4. And, most importantly, it is a manner of allowing and stimulating the students to take command on their own work. They normally try to produce as least as possible to just comply with the teachers' requirement. And when they hand their code over, they most often remain in distress, awaiting the surely long list of failures arising from the test to which it is going to be submitted. We try to call their attention to such a disgraceful fact: How come they do not know how *their own work* behaves? Orderly methods of reasoning and of development are appreciated as they show practically useful and allow to take full command on one's own doing.

Finally, as to the difficulty inherent to Programming and the practical value of the various flavours in which the latter may be offered, we have nothing to add to several well-known quotes by e.g. Dijkstra.

1.2 The approach to Mathematics

On the other hand, the mathematical education of our students is defective in several senses:

1. They do not link the practice of Mathematics with that of proof. They rather believe Mathematics is about (rules of) calculation.
2. Their education stresses Mathematics of the continuum, as a consequence of programs of study that are shaped in accordance to education in classical Engineering branches. So they believe Mathematics is about calculation of roots, limits, derivatives, integrals and the like.
3. They do not link proof with *understanding*. When faced with the necessity of producing one, they are exclusively preoccupied with how a proof is to be *redacted*, as if proof would arise from the correct observance of a liturgy, which is to be learnt by heart.
4. They are not illustrated in the identification and application of methods or strategies of proof and problem solving generally. For instance, in a typical high-school Mathematics course, they are exposed to theory lectures, which might or might not include some theorem proving, after which they receive sizeable lists of exercises which they are supposed to solve entirely on their own. The exercises are indeed applications of the theory, but some of them could be rather involved, and many others respond to patterns or strategies of approach. Unfortunately, no discussion arises normally at the level of reflection about the (methods of) solution encountered. A regrettable consequence is that the students believe that strategies or disciplines do not exist in Mathematics or problem solving, and that everything therein is a matter of wit, which they perhaps happen not to possess. In short, they are rarely taught to *think*, systematically.
5. They fail to identify any significance of Mathematics for their software engineering education or practice. This is indeed an immediate consequence of the approach to Programming which they receive as natural, as commented above. But it is also due to the conception of Mathematics that they make up for themselves as a consequence of all the preceding facts.

We are certainly in need to face and revert such conception. Our contribution is described in the following two sections: First we present the general ideas of a course intended to teach a mathematical approach to programming, next to which we describe in detail the syllabus of the course. In the last two sections we assess the experience of having the course taught for the first time and expose final conclusions and further work. The appendix includes samples of the definitions and proofs given in the course.

2 An approach to Mathematics and Programming

The way out that we have managed to articulate is the setting up of a course of Mathematics, whose objects happen to be given by a programming language and can easily be processed by computers. In other words, we will be doing Mathematics: As in conventional practice, we shall employ natural language (in our case, Spanish) equipped with some notation designed to represent the various objects under study. In our case, such notation will be a programming language. Therefore we will be able to learn the language, as well as methods of program construction and of proof of properties of the programs constructed. As already said, the programming language in question should be *functional*. But we have also other requirements, namely:

1. It should be *concise*. More specifically: it must be possible to describe it entirely with utter brevity, say in one page of text, so that as a consequence the students are able to keep it in mind in its entirety at all times. Also its various constructs should be as orthogonal as possible, so that ideally there is exactly one way to encode every relevant programming mechanism.
2. It should be (simply) *typed*. We believe that the discipline of types is quite helpful for learning and carrying out methodic, safe programming. On the other hand, the use of simple types in a Hindley-Milner system introduces the very convenient and conceptually rich notion of parametric polymorphism.
3. It should allow to define *inductive types*.
4. Syntax should be flexible so that we do not jam ourselves in the course of the development.

On these premises we chose to formulate a λ-calculus close to the core of Haskell, with a generous supply of symbols and fixity combinations thereof. In short, it is a minimal typed functional language, easily translatable into Haskell. As a perhaps remarkable peculiarity, we do not use pattern matching: functions are defined by giving a name to a λ-abstraction and case analysis and recursion are encoded by employing only the case construct.

The features separating this language from the ML family are the mechanism of *type classes* as well as *lazy evaluation*, but it has to be said that none of them is to our current opinion essential for our purposes. Indeed, regarding type classes, what we are interested in having is rather a mechanism as simple as possible for introducing *abstract structures*, of which the overloading polymorphism is a particular case. Type classes are rather restricted for achieving a natural representation of arbitrary abstract structures mainly because it is awkward to use the same concrete type in more than one implementation of the operations of a type class[2]. We therefore could, but have not yet tried to, use a module system for this purpose, either the one of Haskell or that of ML. On the other hand, regarding lazy evaluation, it gives us direct access to programming infinite structures but we have found it too demanding for first-year students to fully introduce and employ the methods for reasoning about them.

The contents of the course are foreseeable given the decisions just enumerated: We start by introducing the fundamental notion of *type*. Next we treat *functions*, i.e. introduce the functional types. After that we start introducing ground data with the *inductive* types: First the ones with finitely many objects, of which the *booleans* are of paramount importance, and next those with infinitely many objects, of which the simplest one is the type of *natural numbers*. Then we just go on to consider *lists* and different kinds of *trees*, including e.g. languages. Therefore, as a course of Mathematics, ours corresponds roughly to Arithmetic enhanced with some Computing Science topics, like some theory of lists and trees. It starts

[2]For instance, for implementing tables or dictionaries using lists both ordered and unordered.

from scratch, i.e. introducing by way of explanations its primitive notions, and we set ourselves as goals to reach:

1. The Fundamental Theorem of Arithmetic,
2. the correctness of at least one List Sorting algorithm, and
3. programming an interpreter of a small programming language.

We also assume the purpose of explicitly reflecting about strategies of proof and problem solving.

A noticeable feature of the Mathematics of this course is that it does *not* start with the notion of *set*, nor it takes as given the e.g. integer numbers. We rather start with the notion of type which is, as a matter of fact, essentially syntactic in character, since the fact that an expression has a given type is mechanically decidable. This corresponds to the simple fact that the well-formation of an expression in a mathematical text must be decidable just by reading, i.e. it must not require any mathematics. The first mathematical notion is therefore that of (computable, higher-order and polymorphic) *function*. And the subsequent notion is that of *inductive type*, i.e. the several forms of *trees*, including the very simple ones which consist of only one node (i.e. the types with finitely many objects) or are linear, either with or without information at each node (lists and natural numbers, respectively). Inductive types are defined uniformly as those types given by *enumeration* of their *constructors*, which are of course generally functions. Such definition is equivalent to introducing the primitive recursion and induction principles for each inductive type. To the two notions of function and tree we add only that of *abstract structure*, which is given as an aggregate of data types and functions with determinate properties[3]. In the course, we employ abstract structures for implementing overloading and for illustrating how to introduce abstract algebras and abstract data types but do not pursue their study or exploitation. We *do* emphasize the remarkable economy of concepts, which seems to us an interesting and pleasing finding arrived at as a consequence of pursuing this particular development: all that there is is functions, trees and abstract structures.

As has been observed, all our functions are by definition computable. We do not restrict ourselves a priori to constructive methods of reasoning, but have had no necessity to employ indirect proofs. So, the Mathematics of the course is, as a matter of fact, constructive.

In view of all the preceding issues, the many very good available textbooks on Functional Programming are not quite adequate to our purposes, mainly because of our preference for the economy of concepts already noticed as well as for an order of presentation that is as close as possible to the logical one given by the definitions of the various concepts. Also there is difficulty in finding a course of Mathematics somehow along the desired lines. Closest is *The Haskell Road to Logic, Maths and Programming*, by Doets and van Eijck[3], which treats very similar contents. But they base themselves on Set Theory as is customary in Mathematics and, more importantly, the Haskell code is not there as the notation for the objects on which the mathematics runs, but only as a complement, useful to learn functional programming and because of the additional insight provided by the design of the algorithms that represent interesting mathematical constructions. For us, in contrast, it is essential that the mathematical objects are the programs to be constructed and that therefore the theorems are about relevant properties of precisely those programs. In other words, we intend not to teach Mathematics *and* Programming, separately or in parallel, but only one subject, viz. the mathematical presentation of Programming. We also assign some importance to pursuing the alternative foundation on types and functions.

[3] Abstract structures are of course *record*-like, but we do not enter into such considerations in the course.

The latter places our conception quite close to the one sustaining the work in Constructive Type Theory, e.g. of the Mathematics that has been developed in systems like Coq[5] and Agda[7], and probably our primal reference should be the work by Robert Constable [1]. It is, however, important to point out firstly that such conceptual vicinity has not been sought deliberately. Rather our intention was from the beginning the one that we have been exposing, namely to reason about programs —i.e. to formulate the Mathematics of programs— in as direct and concise a manner as possible. It is this purpose that has conducted us to the present approach, particularly to taking the programming language concepts —e.g. the concept of function— as the primitive or real ones, not depending on the introduction of any other kind of mathematical object. On the other hand we of course do not employ Type Theory, which brings about several differences with the mentioned works: First, Constructive Type Theory in any of its variants is a fully formal system, whereas we employ just natural language equipped with programming notation and do not make explicit the forms of reasoning employed. We consider it simply excessive to expose our students so early to the severity of a fully formal language of Mathematics. Secondly, the logic of Constructive Type Theory is internal, i.e. a part of the programming language, by way of the isomorphism between propositions and types. We do not make use of such correspondence, and our mathematics runs entirely up and above the programming language. As a consequence, for instance, we are able to permit unrestricted recursion and thus non-termination. In retrospect, it might be doubtful whether this is a feature or a bug, but it certainly facilitates the design and study of several interesting and important algorithms, besides of course allowing (requiring) the study of methods for proving well-foundedness of general recursion.

3 Syllabus

The following is a summary of the syllabus of *Foundations of Computing*, as written just previously to its first instance in the second semester of 2013. The full version includes a detailed description on the method of teaching, which we omit here because it has already been commented on. We do not just list the subjects of the course, but try to give detailed explanation of the specific contents of each chapter as well as of the way they are presented. In the appendix we include samples of the definitions and proofs given in the course.

3.1 Learning objectives:

1. *Mastery* of what may be called *basic Functional Programming*:
   (a) Definition of inductive types on demand.
   (b) Recursion: structural as well as general well founded.
   (c) Use of polymorphic higher-order functions.

   Mastery means they must able to describe the concepts and methods involved, they must know how to apply them, and they must be able to explain *why* they work. They will program functions to perform moderately complex transformations.

   This objective is *mandatory*, i.e. no course approval is possible on its failure.
2. To formulate sound arguments that make evident demanded properties of programs. This entails:
   (a) understanding *why* the desired property indeed holds,

(b) communicating such idea,

(c) employing for the latter some procedures in a systematic manner.

As to this objective, the student should acquire reasonable proficiency. In particular in the case of proofs by induction it is mandatory that she formulates correctly every base case and induction step.

3. They must be able to describe the idea of *abstract structure*, knowing how it can be used to model:
   (a) overloading polymorphism,
   (b) abstract algebras.

3.2 Syllabus and method

Time load. 5 lesson hours per week during 16 weeks = 80 hours, including examination.
Subjects.
0. Introduction and motivation (2h). Where we mainly comment on the bad shape of current Software Engineering practice, especially from the point of view of the quality of software generally. We also describe the general idea of the course, i.e. to do Mathematics on objects that can easily be encoded and processed by computer.
1. Types, objects and expressions (3h). Where we start introducing our programming language (call it λ_{h}) by explaining the idea of *type* of data or objects, and of *expression*, which, to be meaningful, must be a type expression or have some type. We also explain how to write down *definitions*, either of types or of expressions of a given type. We stress that everything that can be written down in the language is:

- *assertions* of the forms α ***type*** or $a :: \alpha$, and
- *definitions* of types or expressions, each of them written generally as an equation $D = d$ preceded by a *declaration*, either D ***type*** or $D :: \alpha$.

2. Functions (5h). Where we introduce the *function types* (i.e. the corresponding axiom or rule of formation), characterizing functions by the operation of *application* (i.e. we give the corresponding typing rule.) Next we go on to introducing the use of *variables* in the language, which can be done simply by resorting to universal quantification of the assertions $a :: \alpha$. With basis on such notation, we introduce λ *abstraction* and the β rule. Also we discuss *bound* variables and the α rule of definitional identity.
3. Type parameters, polymorphism and higher-order functions (5h). Type parameters are introduced by universally quantifying on types the assertions of the language. This allows to form several examples of polymorphic higher-order functions. We also introduce the ideas of type checking and type inference. We discuss proofs of identity: Computations as particular cases and the general proofs of identity. Many examples and exercises are given.
4. Types with finitely many objects (10h). The *empty* type —Non-termination: *bottom* as an expression having every possible type (in particular the empty one). It is allowed because we cannot *formally* restrict recursion so that it is necessarily terminating. *Strictness and laziness*: the method of evaluation of the expressions so far introduced. The *unit* type: notion of *constructor*. *Bool* —the case construct, its evaluation and derived identities. Definition of functions by equations of the form: $f = \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ldots..$.The boolean operators and their properties: proof by case analysis. Some other examples like, e.g. *WeekDays*.

5. Equality, Overloading, Abstract Structures (5h). The different relations called "equality": The *definitional* equality, used in the meta-language. Extensional and intensional equality of functions. The *boolean equality*, a function programmed within the language. Polymorphism: uniform or parametric vs. ambiguous or overloading. How to introduce overloading: abstract structures. Examples from algebra.
6. The natural numbers (20h). How to form inductive types with infinitely many objects: the use of *function constructors* operating on the same type, which yields recursively formed objects. The natural numbers as the simplest type with infinitely many objects. Its case construct. *Recursion*: primitive. *Induction*: primitive or mathematical. *Partial functions:* Pre-conditions. *Contracts:* Post-conditions. *Structural* recursion and corresponding induction principle. *Well-founded general* recursion and induction. Proofs of termination. *Divisibility and primes.*
7. Lists (15h). Definition, case construct, primitive-structural recursion and induction. Higher-order functions: *map*, *filter*, *foldr*, etc. Well-founded general recursion and induction. Ordered structures and *Sorting*. Infinite lists. *The Fundamental Theorem of Arithmetic.*
8. Trees (15h). Every inductive type is a type of trees—so there are *only* functions, trees and abstract structures. Binary trees, with order. *Languages:* Interpretation-embedding.
Evaluation. There are two written tests. The first one is at the end of the chapter on Natural Numbers, the second at the end of the course. Altogether they sum up 85% of the evaluation. The other 15% is achieved by laboratory work (i.e. production of running Haskell programs). The course is fully approved by obtaining at least 86% of the points. With a score between 70 and 85% the student passes to a further instance of examination. With less than 70% she has to retake the course.

4 Assessment

We start by describing some peculiarities of the only instance of the course that has been completed, namely the one corresponding to the second semester of 2013. The most important one is that we were not able to employ a method of teaching based on previous reading of the corresponding material by the students followed by work on problems in the classroom. The reason for this was simply that we were producing the course notes as the course was progressing and they were often not available in time. This inconvenience is thus easily remedied for future instances, but it has to be said that it is not the only obstacle to proceeding as we intended to. Indeed the main problem as to the implementation of such method of teaching is the reluctance of the students generally to:

1. Invest in learning beyond what is strictly demanded and has score value for approval of the course, and
2. read, in general. And still less, mathematical text.

Now we would like to argue that the way out should *not* be the one accepted and promoted by many, namely to acknowledge that a kind of cultural shift or something similar has happened and we should not anymore ask young people to *read*. We also refuse to rewrite or edit our text so that it becomes somewhat more appealing —in the limit possibly transforming it into a video-clip or something alike. Being able and trained in reading texts of often difficult apprehension is a *plainly indispensable* competence for anyone engaged in complex intellectual activities, as are many others, like the capacity to think for long on a difficult problem. Such activities demand patience and self-acquiescence which are qualities that need to be trained. It is a huge mistake to induce future engineers to believe that every learning or

problem-solving instance should be of immediate elucidation. The prize to be paid is that they only will be able to solve easy problems. So what we propose in this matter is generally:

1. To search mechanisms that make the students heavily depend on the text.
2. To train them in reading, as much as possible, for instance doing some actual reading in class.

The next peculiarity on which we would like to comment is that the main laboratory work of this instance of the course was the embedding in Haskell of a minimal version of the guarded-command language of Dijkstra. The experience was pleasing for us the teachers, as we were able to ascertain:

1. The simplicity with which various solutions can be developed in Haskell, even restricted to our minimal λ_{h}.
2. That the problem was plainly within the reach of our students. (We did provide some of the necessary functions, but they had to understand the full script in order to provide the missing pieces.)

Moreover, we believe the project was also pleasing for a good number of students, as they learnt that an indeed minimal language, in which they had had less that 15 weeks of training[4] in their first year of study, allowed them to construct an interpreter of another language, one that, in addition, was a memory-assignment-while based one, like they thought (or had been taught) it should be. A side-effect of this project is that we wrote notes about the method of invariant-based programming, deriving a short program from its specification together with evidence of its correctness. This was done without using any kind of formal logic, i.e. just in plain natural mathematical language as we had been doing all the time. Since we of course employed programming constructs that they were accustomed to use (rightly *and* wrongly) they could perceive in a quite concrete way that it is indeed possible to methodically arrive at a solution with plain assurance of its correctness, which was for many of them actually more illuminating that what we had been doing during all the course.

Actually the prime question as to how much the students made out of the course should be: *How much more do they care now about the correctness of their own code?* Unfortunately we have not devised or employed any tool adequate to come up with an answer to this. The impression, however, is that, for it to be fully appreciated and educationally productive, the matter should be promoted to the institutional culture wherein it unfortunately does not belong nowadays. Or, in other words, the general culture is that it is generally worthless, or of low impact, to interest oneself in well founded methods of Programming. We could say that the culture has changed if, for instance, invariant-based methods started to be employed in the initial Programming courses. But doing it well requires certainly to abandon Java, which seems to be unthinkable, given the current position of this language in the market and the pitiful fact that we are in a situation in which university education imposes itself the duty to serve the market's demands.

Possibly the next question to ask is *how well did the students?* and the answer is that they did indeed well enough: from a total of 125 students, *51.2%* fully passed the course and *23.2%* had to a take a final instance of examination. To gauge the persistence of the learning in a longer term, we decided to test those students that had obtained a grade of at least *70%* and were enrolled in the subsequent course of *Logic*. This happened 4 months after the end of *Foundations of Computing*, a period in which they had no academic activity (it was the Summer break). The test was anonymous and non compulsory, which means that we relied on the good will of our students to complete it and do so to the best of their ability. It was a written test including problems covering all the subjects considered important, ranging from

[4] As a matter of fact the right number is 12 weeks.

rather elementary problems on booleans and natural numbers to other more elaborated ones dealing with more complex and less obvious trees. We could test 54 students (about half of the students that enrolled in the course originally). From this we gather:

- *38%* showed themselves to master and understand all the concepts pointed out as learning objectives.
- As to mistakes concerning *recursion*: *33%* had trouble determining the appropriate scheme of recursion for the complex inductive types, i.e. as opposed to natural numbers. Most of the errors would have been detected by a standard compiler. *7%* had trouble with primitive recursion over natural numbers.
- As to mistakes concerning *proofs by induction*: *18%* omitted an induction step while proving and *16%* made some other error while proving. These two are related since it is usual for a student to hesitate and erase an induction step instead of presenting it containing errors.

A possibly more informative comment in this respect is:

1. The most difficult matter of the course for the students was to properly realize the link between *inductive definition*, *structural recursion* and *structural induction*. The difficulty starts of course with the use of recursion, but that is surpassed relatively rapidly. What remains difficult for them to assimilate is the formulation, validity and use of the principles of induction. This is rather surprising, as one conceives this matter as lying very close to the former, so that understanding induction should not take much longer than understanding recursion. The reason seems to us to lay again on the poor mathematical education that they carry.
2. In connection to the mentioned difficulties, we can say that they are doing well *now*, i.e. in the related courses of the subsequent semester. Specifically, we are teaching the same students the new course of *Logic* which was moved forward as a consequence of the creation of *Foundations of Computing*. *Logic* has now more room for the classical subjects of Propositional and First-Order Logic, but also can make profit of the functional programming capacity of the students, so that it includes now several programming problems and laboratory projects, mainly related to (semi-) automated methods of proof. We of course make use of recursion and induction —viz. on the formulas of the logical languages at hand— and can tell that, in this second look at the issue, all of the students are plainly in command of both methods —several of them having really understood the way induction works only now.

We would like now to say something about the results of the various design decisions concerning this course. The first thing is that our most modest initial intention, namely to give induction and recursion more room in the education of our students, is showing itself correct, since we have no problems with such matters in our *Logic* course. Moreover, the students are perceiving that fact and therefore the usefulness of the preceding course. But of course we have also attempted to introduce functional programming as a mathematical activity. As already commented, we believe that the mathematical nature of programming is not being successfully incorporated, as the general culture persists in eluding the issue. Functional programming, however, has been given a birth within our program of studies and its prospects are good. We already use it in a subsequent course, namely *Logic*, and have the opportunity to continue with its application in two future obligatory courses on Theory and Technology of Programming Languages, so that we can expect our future engineers to have a satisfactory proficiency

in functional programming, especially concerning its very successful application to language processing and meta-programming.

What about the specific language we have chosen to employ? Well, our evaluation is that the choice has been a plain success. First, it has proven itself an able tool for comfortably composing all the programming of the course, which was not at all trivial. Secondly, it reflects quite precisely the conceptual structure of the course: All that exists is functions, inductive types and abstract structures, and in λ_{h} they are treated as follows:

1. There is but one way to define functions, and it is very direct: you give a name to a λ-abstraction.
2. Inductive types are defined by enumeration of their constructions: For introducing inductive types we use Haskell's `data` declarations with full typing of the constructors and explain that it is a kind of packaging notation for assertions of the primitive forms, i.e. α ***type*** and $a :: \alpha$. Immediately associated to each inductive type we have a `case` construct which constitutes the standard manner of exploiting data of the type in question. Further, on the basis of the constructors the mechanisms of structural recursion and induction are immediately formulated. As to general recursion, it demands consideration on a case-by-case basis.
3. We have certain difficulty only with the notation for abstract structures, which needs to be elaborated a bit more. We have nevertheless settled in practice for a notation close to that of Haskell's type classes since, as a matter of fact, it is enough for the programming we do in the course.

Thus the language has remained small, orthogonal and plainly coherent with the concepts learnt. In particular, the use of `case` instead of pattern-matching proved itself a successful decision, since it makes it absolutely clear what is meant by *proceeding by recursion on this or that argument*. The construct serves also well for encoding what otherwise would be local definitions, which are in fact not necessary. We have only found some abbreviatory use to the employment of ordered pair patterns directly in lambda abstractions. We have also devised a notation for function specifications, called *contracts*, which consists in declaring a pre- and a post-condition. The first is a predicate or relation on the nominal arguments of the function and the second a relation between these and the result. In the course we have mainly used the pre-conditions, in order to specify partial functions. Contracts can easily be embedded into Haskell functions so that they are checked at run-time.

What now about the particular order of presentation of the concepts in the course? We have chosen to follow the logical order of definition. Thus, the explanation of the notion of type precedes that of object, and functions need to be introduced in advance of any ground data, since the latter belong in the general notion of inductive type and this generally depends on functions in order to fully define the notion of constructor. To us, this is a very important feature, granting that nothing is used before being defined in plain generality. This should provide soundness to the whole development and certainty to the student about what can be used and in what manner. The price to be paid is that, for instance, during the first three weeks of the course they do not know of any official ground data with which to compute and often become rather anxious about where the entire thing is leading to. Some of us think this is a problem, while others think it is a nice feature —one that gives the opportunity to speak about the machinery of Mathematics at a level the students did never and probably shall never see anywhere else.

Finally, the proofs produced look generally quite close to *formal* ones, i.e. to proofs carried out in a formal system of Logic, as can be attested by taking a look at the appendix. The reason for this is our starting from scratch and building up just from recursive definitions of the most basic operations. This poses a challenge as to the fluency and abstractness of the entire development which is probably the most serious aspect of didactic investigation that we face concerning the whole endeavour. The students, however, willingly realize that proofs ought not to be reduced to minute steps and rapidly devise more

or less adequate means of expressing them succinctly. Nevertheless, it remains highly interesting to develop an elegant and systematic proof style, for which we are currently looking at the calculational mathematics of Dijkstra and others [2].

5 Conclusions and further work

We have designed and taught a course, called *Foundations of Computing*, in which (Functional) Programming is presented as Mathematics. Concretely, we designed a very concise simply typed λ-calculus with inductive definitions and record-like structures (called λ_{h}) and articulated a presentation and exploitation of it which amounts to a course of Mathematics whose objects are the types and expressions of that language and therefore the theorems state properties of things that happen to be programs.

We claim that this is a serious educational approach to Programming, as it makes explicit the need and means to ensure correctness of programs by means of *proof*, i.e. of production of the required evidence. To put it boldly, we like to say that *a program is a theorem*. We also claim that it is an interesting and rich approach to Mathematics, as it discloses many aspects of its workings that are not scrutinized in more conventional courses. Also it is Mathematics of very concrete objects that can be used on computer. In its present scope it encompasses Arithmetic plus some theory of lists and trees belonging in Computing Science, it is founded upon the notion of computable function instead of on that of set and, as a matter of fact, it is constructive. When regarded as an attempt to take seriously the traditional claim that functional programs are mathematical in nature, our work can rightly be said to be a positive comprobation thereof, since the entire development is sound and pleasantly harmonious.

We have taught the course to first-year students of a Software Engineering program, which has allowed us to ascertain the adequacy of the language and the general approach chosen. The students have generally acquired decent proficiency in basic Functional Programming, as attested by their continued use in a subsequent course. Also they have incorporated an acceptable command on proofs by induction. On the other hand, we believe that a cultural transformation of some larger order is necessary for them to be willing to systematize their programming habits in the direction of methods that take correctness assurance seriously. We also believe that the learning outcome should and can be improved in future instances, particularly by favouring autonomous study and more extensive, collective discussion of problems.

Some interesting lines of work arising now seem to be connected with the logico-mathematical aspects of λ_{h}: First, we would like to complete a presentation of the language starting off from the meaning explanations of the notions of type, expression of a type, variables, etc., i.e. very much in the form of Martin-Löf's foundations of Constructive Type Theory[4]. The difference as to this is of course that our language is simply instead of dependently typed and does not incorporate the propositions-as-types principle, since our Mathematics runs up and above the language. As a consequence, we also allow unrestricted recursion and therefore non-termination. Since our language is very close to the core of Haskell, these meaning explanations could be taken as providing a direct foundation to Haskell too. These founding explanations can take different forms, of which some are preferable from a logical point of view while others are more didactic.

We also would like to compare our Mathematics with that of standard textbooks on the same subjects, especially concerning the level of detail and the constructive character of the proofs. Further, we must investigate the possibility of formulating the real numbers in λ_{h}. The need for a clean treatment of infinite objects prompts the investigation of adequate linguistic features. We believe that a formulation in terms of functions, inductive types (unions) and record types (products) will eventually prove itself

most adequate for fulfilling all the programming and mathematical requirements to be met.

Concerning didactics, the most interesting aspect to investigate is the development of a systematic and elegant proof style that reconciles the very concrete and detailed level at which definitions are formulated with a pleasant fluency of the discourse. For this, the use of the methods of calculational mathematics of Dijkstra and others [2] should be pursued extensively and adapted. It is also a fact that the whole content of the course is not far from being able to be actually formalized by transporting it into a proof assistant like e.g. Coq [5] or Isabelle-HOL [6]. It would be nice to make the experience of actually having the course worked out at that level, specifically investigating the formulation of tactics that could be plainly accessible to undergraduate students.

Finally, we are also initiating an experience of teaching this material to senior high-school students. This is interesting because we believe that the material is indeed within their reach and it could be fruitful concerning the prospects of massive education in Mathematics and Programming.

References

[1] R. L. Constable (1984): *Mathematics as Programming*. *LNCS* 164, pp. 116–128.

[2] Antonetta J. M. Gasteren (1990): *On the Shape of Mathematical Arguments*. Springer-Verlag New York, Inc., New York, NY, USA.

[3] Kees Doets & Jan van Eijck (2004): *The Haskell Road to Logic, Math and Programming*. King's College Publications.

[4] P. Martin-Löf & G. Sambin (1984): *Intuitionistic type theory*. Studies in proof theory, Bibliopolis.

[5] The Coq development team (2004): *The Coq proof assistant reference manual*. LogiCal Project. Available at `http://coq.inria.fr`. Version 8.0.

[6] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[7] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, University of Gothenburg.

Appendix: Sample Proofs

Integer division

▷*The booleans:* data $Bool = \{False :: Bool, True :: Bool\}$.
▷*Natural numbers:* data $\mathbb{N} = \{0 :: \mathbb{N}, S :: \mathbb{N}\to\mathbb{N}\}$.
▷*Sum and Product:* Defined as usual by primitive recursion. Proven associative and commutative by straightforward inductions.
▷*Subtraction:* Requires the *predecessor* function. We write full definitions to illustrate the specification of preconditions.

```
pred :: ℕ→ℕ
--{pre on pred n : n ≠ 0} (precondition)
pred = λn. case  n   of {
                 0   →   !!! ;
                 Sx  →   x
           }.
-- !!! is formally ⊥ and is encoded in Haskell as some appropriately complaining ill-terminating evaluation. It
would be also acceptable to just ignore the case, but it is better practice to program it explicitly.
```

Propositions in the mathematical language are *not* boolean expressions of the programming language. One reason for this is that propositions must never be undefined, whereas boolean expressions might. Another, possibly more important, reason is that predicates and relations should not be defined by programs, but rather play the role of specifications prior and above the writing of programs. The prime mathematical relation is $=$ which is the equality induced by the definitions formulated in the programming language (i.e. it is the definitional equality). The specification (pre-condition) of subtraction makes use of the usual order relation $\leq$ on $\mathbb{N}$, which can be defined as follows:
$m \leq n \ \equiv \ (\exists k :: \mathbb{N})\ n = m + k.$
Similarly, the strict order relation is defined by:
$m < n \ \equiv \ (\exists k :: \mathbb{N})\ n = m + Sk.$

$(-) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
--{**pre on** $m - n$ **:** $n \leq m$}
$(-) = \lambda m.\lambda n.$ case n of {
$0 \ \to \ m$;
$Sx \ \to \ pred(m - x)$
}.

▷*Some lemmas:* The following are proven readily:
Lemma 1. $Sm * n = m * n + n.$
Lemma 2. $n \neq 0 \Rightarrow \ m - n < m.$
Lemma 3. $n \leq m \Rightarrow \ (m - n) + n = 0.$
▷*Boolean order* <= *on* $\mathbb{N}$*:* It is defined within an instantiation of the structure[5] *Ord* with $\mathbb{N}$. It is proven to be an implementation of the order relation $\leq$, i.e. $m <= n = True \ \Leftrightarrow \ m \leq n$. We omit the details for brevity.
▷*Quotient and remainder:*

$(\div) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
--{**pre on** $m \div n$ **:** $n \neq 0$}
$(\div) = \lambda m.\lambda n.$ case $n <= m$ of {
$False \ \to \ 0$;
$True \ \to \ S((m - n) \div n)$
}.

$(\%) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
--{**pre on** $m \% n$ **:** $n \neq 0$}
$(\%) = \lambda m.\lambda n.$ case $n <= m$ of {
$False \ \to \ m$;
$True \ \to \ (m - n) \% n$
}.

The recursion is well founded in both cases, as *Lemma 2* is applicable. (This is fair enough as a proof of termination.)
▷*The correctness properties.* The following results are straightforward:

Lemma 4.

1. $m < n \ \Rightarrow \ m \div n = 0.$
2. $n \leq m \ \Rightarrow \ m \div n = S((m - n) \div n).$
3. $m < n \ \Rightarrow \ m \% n = m.$
4. $n \leq m \ \Rightarrow \ m \% n = (m - n) \% n.$

[5] I.e. *type class* in Haskell.

These lemmas are the "pattern-matching" rewrite of $(\div)$ and $(\%)$. The same kind of lemmas is normally to be stated and proven for every recursive function to be treated. In this case they rely on the correct behavior of `<=`.

First correctness property of quotient and remainder:
$n \neq 0 \;\Rightarrow\; m = (m \div n) * n + m\%n$.
Proof by well-founded induction on m:
$\underline{m < n}$:
$(m \div n) * n + m\%n$
$=$ ($m \div n = 0$ and $m\%n = m$ as $m < n$, using *Lemma 4*(1 and 3))
m.
$\underline{n \leq m}$: The induction hypothesis (well-founded induction) is:
$(\forall x < m)\ (n \neq 0 \;\Rightarrow\; x = (x \div n) * n + x\%n)$.
$(m \div n) * n + m\%n$
$=$ (*Lemma 4*(2 and 4))
$(S((m-n) \div n)) * n + (m-n)\,\%\, n$
$=$ (*Lemma 1* plus term reordering)
$((m-n) \div n) * n + (m-n)\,\%\, n + n$
$=$ (induction hypothesis, as $m - n < m$ since $n \neq 0$)
$m - n + n$
$=$ (*Lemma 3*, as $n \leq m$)
m.

Second correctness property of quotient and remainder:
$n \neq 0 \;\Rightarrow\; m\%n < n$.
Proof by well-founded induction on m:
$\underline{m < n}$:
$m\%n$
$=$ (*Lemma 4*(3))
m
$<$ (hypothesis)
n.
$\underline{n \leq m}$: The induction hypothesis is:
$(\forall x < m)\ (n \neq 0 \;\Rightarrow\; x\%n < n)$.
$m\%n$
$=$ (*Lemma 4*(4))
$(m-n)\,\%\, n$
$<$ (induction hypothesis, as $m - n < m$ since $n \neq 0$)
n.