Six ways to understand monads

Experiences from the functional programming course at TU Delft

Erik Meijer and Georgios Gousios
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
Email: {h.j.m.meijer, g.gousios}@tudelft.nl

Abstract—**In the last few years, the popularity of functional programming as a way of solving computational problems has increased significantly. While most computer science curricula do include a course on functional programming, in many cases it is disconnected from practical applications, which is precisely where functional programming shines. To fill in this gap, we designed a functional programming course that required students to learn by experience with real world applications. In this paper, we present the course's design and outline our experience from delivering it.**

Index Terms—**functional programming, teaching**

I. Introduction

Due to a variety of reasons, including the advent of cloud computing, the rising rate of information production and the necessity to reach the market fast, currently, large corporations and start-ups alike are investigating alternative programming and information storage models. As a result, during the last few years, the practical software engineering field is witnessing a noticeable shift towards functional programming. Scripting languages, notably Javascript and Ruby, pioneered the introduction of functional concepts, such as closures and lambda functions, to mainstream programming. A new wave of programming languages, developed to overcome the expressiveness and complexity limitations exhibited in mainstream languages, have promoted functional constructs, such as type safe pattern matching, higher order functions and single assignment variables, to first class citizens (Scala). New, purely functional, languages have emerged to fill in the remaining gaps (F#, Clojure), often introducing significant advancements in their field of specialisation (such as Erlang in distributed fault-tolerant systems). Finally, large scale information processing systems such as Map/Reduce [1] and domain specific languages such as LINQ have integrated functional concepts to ease the expression of computations.

Broadly speaking, functional programming is a style of programming in which the primary method of computation is the application of functions to arguments. Among other features, functional languages offer a compact notation for writing programs, powerful abstraction methods for structuring them, and a simple mathematical basis that supports reasoning. Many of the advanced techniques in modern functional languages, such as monads and catamorphisms, are closely based on principles from category theory such as functors, initial algebras, monads and Kleisli categories.

While functional programming has been taught for long in computer science departments [2], curricula tend to emphasize functional programming theory rather than practical applications. Special programming languages are used to teach functional programming specific concepts, while little connection is made to how those concepts can be transfered to non-functional languages. The course at TU Delft attempted to teach functional programming principles in the classroom and then expect students to apply those concepts in the (non-purely functional) language of their choice. The course had a very strong teaching by example focus: students were expected to participate in both in-classroom exercises, homework assignments and implement a real world system as a final project.

II. Challenges

While planning the course, we dealt with the following challenges related to the course's organization and content.

A. Heterogeneous background of participants

The functional programming course is elective at the MSc level. Students from all departments in the Electrical Engineering, Mathematics and Computer science faculty are allowed to participate. For practical reasons, the enrollment to the course was limited to 15 participants. This permitted us to get to know the students individually and to provide them with personalized support.

From the students that enrolled, most have received formal introduction to imperative and object oriented programming in their bachelors curriculum, while through participation to other courses they had limited exposure to functional programming (i.e. Map/Reduce in the web information systems course). Two of the students were majoring in computer engineering, which meant that their programming experience was restricted. Several students also had work experience as programmers as part of their industrial placement or through participation to start up ventures.

The diverse backgrounds of the students guaranteed that no generic introduction to the topic would be sufficient. For this reason, we decided to adopt a hands-on-first approach; the students would have to learn by flexing their programming muscles, instead of being introduced to the theory through toy examples.

B. Choosing the appropriate topics

Functional programming is arguably the oldest programming paradigm. In it is purest form, it is based on a minimalistic theoretical background (λ-calculus [3]). Relatively recent formulations [4], [5] also introduced concepts from category theory. While the every day use of functional programming does not necessarily require the programmer to be aware of the theory, understanding it usually leads to more elegant algorithmic solutions. Functional programming teaching is also associated with advanced type systems; indeed, the flagship functional programming languages (Haskell and the ML family) both offer very sophisticated support for types. Consequently, teaching the full spectrum of functional programming theory and techniques would have been impossible in a half-semester course; instead we decided to focus on practical aspects of data processing, transformations and state representation using functional techniques.

C. Choosing the appropriate programming language

Programming languages, in addition to enabling the programmer to express a series of instructions to be executed by a computer, affect the programmer's thought process and consequently her approach towards problem solving [6]. When teaching a programming course, it is important to be able to demonstrate concepts without interference by the chosen language's syntax. However, a non-practical language might have the opposite effect; our experience has shown that the further a demonstration language is from practical application, the less important student feel the taught concepts are. Fortunately, most functional programming concepts can be expressed cleanly by several widespread languages: for example, Javascript, Ruby and C# have closures, first class and higher order functions, while a number of emerging languages, such as Scala, F# and Clojure are functional languages implemented on familiar development platforms. Apart from staying compatible with the existing literature, there is no practical reason that necessitates teaching of functional programming strictly in a language like Haskell or ML.

The above led us to not choose any particular language for the course. During the lectures, the demonstration languages where Haskell and C#, while at the labs we used Scala. We actively encouraged students to use the language of their choice on the platform of their choice to carry out homework assignments and the course's final project.

III. The Course

The high level goal of the course was to teach the principles of functional programming, and the corresponding Category theoretical principles. More specifically, the educational purposes of the course were:

- To introduce students to basic functional programming concepts, higher order functions, monads and advanced type systems.
- To introduce students to the approach of expressing data processing problems as a series of function applications.
- To explain the application of functional concepts in non-purely functional environments.

The course consisted of a series of lectures, of which two where invited lectures by an external instructor, and laboratory sessions. In total, the course consisted of 14 hours of lectures and 8 hours of lab work. The students also had to carry out a series of homework assignments and a final project. The students would be evaluated by their performance on the final assignment. The lectures took place in a quarter (half semester), during 3 intense weeks; the students had another 6 weeks to finish their final projects.

A. Lectures

The lectures covered the following topics:

- Functions and functional composition, higher order functions, recursive functions, avoiding recursion
- Basic types, function types, currying
- Lists, mapping and traversal using folds
- Monads, composition and their application on the LINQ query language [7]
- Monoids, functors and their application on data structure processing
- Functional formalization of the Map/Reduce data processing paradigm [8]

The lecture material was loosely based on Graham Hutton's functional programming course at the University of Nottingham. The course's recommended book was Programming in Haskell by the same author [9].

B. Student projects

At the end of the lecture period, the students were given a selection of projects to work on. The projects included:

- Real time graph visualisation on steaming data. The particular example that students worked on was visualizing community structures for Github projects.
- Multi-source real time data processing. Students worked on a programming language popularity index, based on data from the Github event stream and the StackOverflow tag cloud API.
- Implementation of Haskell constructs in Javascript. Students implemented the Haskell Prelude (basic functions for list manipulation).
- Implementation of Map/Reduce algorithms in Haskell. Students installed and configured cloud Haskell and implemented simple Map/Reduce based algorithms in a distributed setting.
- Implementation of simple constraint solvers.
- Machine learning algorithms. Students implemented Naïve Bayes classifiers and K-means clustering algorithms.

To counterbalance the differences in student experience level, the assignments were open ended; students could drive their projects as far as they where willing to. The deliverables for the assignments were a repository with the source code and short report describing the solution that the students came up with. The students were also required to present a working demo by the end of the course time.

C. Labs

The purpose of the lab sessions where to help students carry out their final assignments and further explain concepts that might not be clear to students. Students convened every week, presented their progress and discussed with each other and with the lab demonstrator design and practical issues that arised in the implementation of their projects. To further motivate students to work on their projects, one of the lab sessions was converted to a full day coding sprint.

IV. Experiences

A. Teaching by example

The focus of teaching was to present functional programming from a practical aspect; we were mostly interested to teach students what they can do rather than the theory behind functional constructs. An important distinction that was made early on was that the world is imperative; therefore while functional programming can be a great tool for thinking about a problem, a pure functional solutions cannot reflect on the real world. As a consequence, it is usually best to mix programming paradigms, i.e. imperative or object-oriented programming for state representation and functional programming for data processing. In our experience, this distinction helped students understand that they are already doing functional programming without noticing; enforcing data immutability to handle multi-threading problems or passing function arguments in scripting languages are applications of functional programming principles. The moment students realized this fact, their attention was captured to the lectures.

During both lectures and labs, all non-whiteboard examples where actually entered in a real programming environment (LinqPad or the Scala command line read-evaluate loop) and the results of the evaluation where discussed with the students. Participation was encouraged by modifying the examples and asking the students to perform the evaluation, before the example was executed. The examples varied from list manipulation with higher-order functions to a toy actor model example.

The proverbial "six ways to understand monads" included presentations of the formal theoretical concepts, visualizations of monads as blenders (based on the monadic property that once a value is captured, it is then contained), applications in Haskell for encapsulating state (through the Maybe monad), applications to generic programming in C#, applications to LINQ and the reactive extensions frameworks, and, finally, a whiteboard exercise where students were prompted to participate in the ad-hoc specification of Haskell's IO monad from basic principles. Once again, the examples enabled the students to understand the composition and state containment properties of an otherwise difficult to comprehend theoretical concept.

B. The teapot exercise

One of the highlights of the taught period of the course was the teapot exercise, which we used it to draw the student's attention to the following facts:

- Most modern programming languages can express functional programming constructs.
- Functional programming works best in data transformation scenarios

The exercise consisted of rendering the Utah Teapot [10] using any graphics primitive of the student's choice using only right triangles. In its core, the exercise required students to decompose arbitrary triangles, which comprised the input Utah Teapot model, to a series right triangles. The students had to come up with the decomposition method (using an analytic geometry method), a decomposition termination criterion to stop the decomposition when triangles are too small to be rendered on screen and a method to recursively apply the above mentioned transformations on the input data. As always, the students could decide the implementation language of their choice. The exercise was given as a mid-week homework assignment, between the Monday and the Friday lectures.

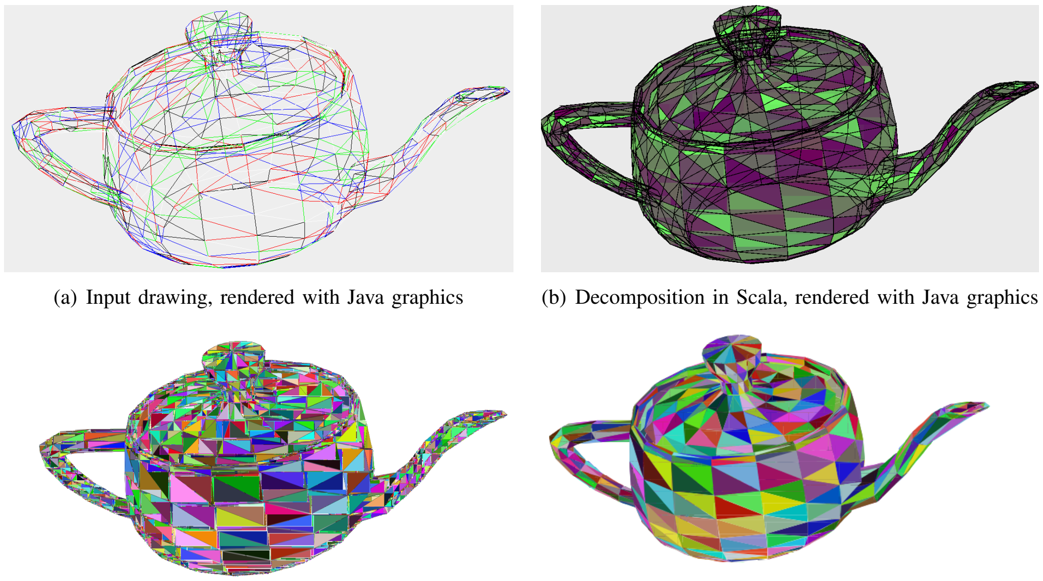
The student's response was overwhelming. Even though it was made clear that the exercise would not contribute to the final grade, the will to apply the data processing techniques taught during the lectures motivated the students to work very hard. The fact that they were instructed to use their favourite language enabled them to focus on decomposing the problem in a series of testable calculation steps rather than meddling with the intricacies of learning a new language. In spite of the advice, students were eager to test-drive the newly acquired knowledge using a functional language; apart from C# and Javascript, solutions were also provided in Scala, Scheme and Haskell. Example renderings produced by student programs can be seen in Figure IV-B.

C. Map/Reduce

According to many students, the Map/Reduce data analysis framework was what drove them to attend the course. The lecture on Map/Reduce featured an invited talk by Prof. Ralf Lämmel, who presented his formulation of Map/Reduce using functional programming principles [8] and explained how those can be applied to process semi-structured data. The talk resonated to students; in their projects, students used implicit, through the selected language's library collection methods, or explicit, by formalizing data processing as a series of in memory Map/Reduce operations, forms of Map/Reduce to process their datasets. Out of personal interest, one student even implemented Lämmel's Map/Reduce formulation in Scala, and through the language's extension mechanisms, adapted it for use as an extension method by any Scala collection type, including parallel collections.

D. The coding sprint day

Coding sprints or "hackathons" have long been used by open source software projects to speed up development and increase participation of community members [11]. In a typical sprint,



(a) Input drawing, rendered with Java graphics

(b) Decomposition in Scala, rendered with Java graphics

(c) Decomposition in Scheme, rendered with appropriately positioned HTML div elements

(d) Decomposition in Javascript, rendered with HTML5 graphics

Fig. 1. Example results from the teapot exercise.

project members set measurable targets and work intensively in a tight deadline to deliver them. To ensure the timely delivery of the student projects, we organized an one day coding sprint, during which the students were requested to i) set explicit targets ii) write code for 7 hours with a short break iii) present a 5 minute demo of their work at the end of the day. All project teams where expected to work in the same room, while communication among the teams was facilitated by the lab demonstrator, to avoid unnecessary distractions.

Interestingly, the coding sprint was organized on student request. The reason was that pressure from other courses did not allow them to be together as teams. The sprint taught students the importance of setting realistic development goals and the necessity of iterative development. Even though all teams were off target at the end of the day, most were able to create a rough prototype in less than 7 hours, using functional programming techniques.

V. Conclusion

Teaching the concepts of solving problems algorithmically can be a daunting task [12]; especially more so, when the subjects are already in a specific mindset. The assumption of the presented course was that imperative and functional programming are really the two sides of the same coin; by focusing on how students can apply rather than just be taught concepts of functional programming using the, typically imperative, languages they already know, allows them to better appreciate the strong points and weaknesses of each paradigm. In our experience, the approach has been successful. Both throughout the initial homework assignments and in their projects, the students demonstrated a high degree of appreciation of functional programming concepts. Above all, we believe that it was the hands-on approach employed in this course that allowed students to sharpen their skills and understand the concepts that were taught during it.

Acknowledgements and Availability

We would like to thank the course participants for their enthusiasm and dedication. The source code to all student projects and the teapot exercise can be found at (or linked from) http://swerl.tudelft.nl/bin/view/Main/FunctionalProgrammingCourse. This work is partially supported by Marie Curie IEF 298930 – SEFUNC.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pp. 137–150, 2004.

[2] S. Joosten, K. Berg, and G. Hoeven, "Teaching functional programming to first-year students," *Journal of Functional Programming*, vol. 3, no. 1, pp. 49–65, 1993.

[3] H. Barendregt, *The lambda calculus: Its syntax and semantics*, vol. 103. North Holland, 1984.

[4] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Functional Programming Languages and Computer Architecture*, pp. 124–144, Springer, 1991.

[5] P. Wadler, "Monads for functional programming," in *Proceedings of the 1992 Marktoberdorf International Summer School*, Springer-Verlag, 1993.

[6] K. E. Iverson, "Notation as a tool of thought," *Communications of the ACM*, vol. 23, Aug 1980.

[7] E. Meijer, "The world according to LINQ," *Commun. ACM*, vol. 54, pp. 45–51, Oct. 2011.

[8] R. Lämmel, "Google's MapReduce Programming Model – Revisited," *Science of Computer Programming*, 2008.

[9] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2007.

[10] A. Torrence, "Martin newell's original teapot," in *ACM SIGGRAPH 2006*, SIGGRAPH '06, (New York, NY, USA), ACM, 2006.

[11] P. Adams and A. Capiluppi, "Bridging the gap between agile and free software approaches: The impact of sprinting," *Multi-Disciplinary Advancement in Open Source Software and Processes*, p. 54, 2011.

[12] G. Futschek, "Algorithmic thinking: The key for understanding computer science," in *Informatics Education – The Bridge between Using and Understanding Computers* (R. Mittermeir, ed.), vol. 4226 of *Lecture Notes in Computer Science*, pp. 159–168, Springer Berlin, 2006.