Learning experience of two students in Formal Methods course with different backgrounds

Ruslan Omirgaliyev and Alen German

[1] Nazarabayev University
[2] {ruslan.omirgaliyev,alen.german} @nu.edu.kz

Abstract. In this paper we report the learning experience of 2 students with different backgrounds. This course was taken at Nazarbayev University during Fall 2019 term. We describe our background related to this course, expectations for the course at the beginning of the semester, motivation for taking the course, the perceived level of difficulty, and perception of the field of formal methods. Additionally, we share some of our thoughts sparked by the learning process.

Keywords: formal methods · learning experience · Maude · Lean · theorem verification

1 Introduction

The background of students drastically affects how they perceive the course and absorb the material. Ideally, explanations given to the students would be in terms that they can already understand, but are not so detailed that major parts of the learning material turn into a review. In other words, the pacing of the instruction would be optimal. Unfortunately, students come from different backgrounds. In this paper, we present the learning experience of the same material from the viewpoint of 2 students with different backgrounds. Alen is a masters student in computer science and Ruslan is a masters student in electrical computer engineering.

This paper covers the learning experience of a formal methods course half-way through the course. So far, the course consisted of 2 parts. The first part covered modeling using functional and rewriting in Maude[1]. It consists of teaching the syntax and rules of Maude and implementations of basic data structures and algorithms in Maude. The second part covered theorem-proof verification in Lean [2]. It includes basics of first-order logic, propositional logic, Curry-Howard correspondence (but not the proof of it), and Lean. The two parts were taught by Antonio Cerone and Hans de Nivelle respectively. At first, Prof. de Nivelle was going to teach about the verification of imperative programs but later changed the topic to Lean.

2 Students' backgrounds

2.1 Alen's background

Before taking formal methods, I had taken a course on first-order logic from the philosophy department and a course on programming paradigms (also taught by Prof. de Nivelle). Programming paradigms covered functional programming and logic-based programming. The functional programming part included the basics of untyped lambda calculus and Haskell. But, my first exposure to functional programming was Lisp in an AI course. We were allowed to use imperative structures in Lisp, but I intentionally tried to stick to functional style to learn to express programs in a functional style. We didn't use any interpreter for lambda calculus. Prof. de Nivelle taught us the rules of the language and we interpreted it by hand. I found the concept currying very neat. I took formal logic as my humanities elective. We were taught to express logical ideas in first-order logic including the usage of quantifiers, wrote proofs, and were introduced to truth tables. We used programs called Tarski's world, Fitch, and Boole [3].

2.2 Ruslan's background

I was not familiar with this course before and didn't even have basic courses related to it. However, I passed Advanced Data Structures and Algorithms course. In this course, I learned some basic programming concepts that were helpful in Formal Methods course. In Data Structures course I got introduced to truth table and logic, which I used later in Formal Method Course. I also took Modeling Simulation course, which familiarized me with the concept of verification and validation of a simulated model. I think these concepts are related to the basics of formal methods.

3 Motivation

The description of the course on registrar:

This course focuses on the use of formal methods in the modelling, specification and analysis of systems. Various kinds of systems from a number of application domains, such as human-computer interaction, software systems, distributed systems, systems biology and ecology, will be considered. Students will learn how to utilize formal notations and automated tools for simulation and analysis.

3.1 Alen's motivation

My motivation for choosing the course is very boring. I had to pick an elective and I chose formal methods by eliminating other options. I found the description of the course on the registrar to be unhelpful.

I wasn't familiar with the concept of formal modeling of systems, so the phrasing sounded very vague to me. Outside of formal methods, "system" is a

very broad term. Even though I didn't understand what the course is about I was sure that I didn't want to take the other electives.

3.2 Ruslan's motivation

When I was choosing this course, my motivation was to discover something new for myself. The description of the course seemed only slightly understandable. As I understood, the goal of this course is to give basic knowledge of how to test program modules, verify any model in software engineering. It seemed to be a good challenge for me as an electrical computer engineering student. So the main motivation for me to take this course was that I would learn to verify software or a model using formal methods. Why do we need formal verification of programs? It is customary to consider the task of specification and verification of programs as some auxiliary activity, an inevitable "appendage" to the main occupation of a programmer — the development of program code. One of the reasons is the increase in the cost of damage due to a missed error in the program.

4 Expectation at the start of the course

4.1 Alen's expectation

As I said in the motivation section I didn't know what formal methods are about. So I didn't know what to expect. I was just hoping that the course would be easy.

4.2 Ruslan's expectation

From the very first day of studying this course, I expected that most of the materials wouldn't be familiar to me, except for the basic concepts that I indicated in the "Background" section. Also, from the very beginning of this course, I must devote extra hours to self-learning new topics. If some materials are not properly digested, then, of course, I could turn to the instructors for help.

5 Learning experience

5.1 Alen's learning experience

In the first week I got introduced to proving imperative programs. It was interesting. Maybe someday I'll have a piece of code that will just beg me to prove its correctness or if proving turns out to be easy enough I would prove pieces of code just as an exercise. By the fourth week, I sort of understood what Maude is. I felt skeptical about the value of modeling human-computer interaction. I motivated myself by looking at Maude as an opportunity to practice functional programming skills.

The material in the lectures and homework was easy. The mathematical definitions in the textbook seemed difficult. I would have to put concentrated effort for a few days just to understand them. I don't find Maude that interesting so I neglected reading the textbook. For now, I am satisfied with having an intuitive understanding of Maude. At this point, I saw formal methods as a curiosity as opposed to something that I find interesting or useful. Therefore, my learning approach was to do the minimum. I listened to lectures, did the homework. But I didn't try to understand the nuances of Maude.

The only difficulty with Maude was debugging. The errors were very tricky to find and the error messages weren't very helpful. If I cared more I would've tried to find how to reduce expressions step by step for debugging purposes. This came back to bite me during the midterm. The midterm consisted of a series of subproblems that built into one program. I made 1 mistake at the start and couldn't find it. As a result, I spent most of the midterm trying to

x that one mistake and eventually gave up and didn't proceed to the following subproblems. Thankfully, Prof. Cerone allowed us to complete the midterm at home and gave me a hint as to where my mistake is.

After covering Maude the material switched to proof verification in Lean.

So far most of the material is familiar to me. First-order logic, lambda calculus. The topics that were new to me are typed lambda calculus and Curry Howard correspondence. The timing of these topics coincided with some frustrating homework from another course I am taking and with midterms. Combined with the perceived ease of these topics I haven't paid much attention to formal methods when we were covering Lean. At the time of writing this paper, I am yet to do exercises in Lean by myself.

5.2 Ruslan's learning experience

During the first two weeks, we began to get acquainted with module checking on Maude. For this reason, we used a book named as Designing Reliable Distributed Systems (A Formal Methods Approach Based on Executable Modeling in Maude), author: Peter Csaba Ölveczky. The number of students during the lessons were four. I think, little number of students is very convenient for us and also for professors. The material for me did not seem so complicated at the beginning. But because by this moment I was engaged in my dissertation, I did not pay enough attention to this course for these 2 weeks. I realized my mistake the following week when I had to do my homework. I immediately had questions related to the condition of the tasks, and a bit of fear that I was doing it inappropriately. Since Maude was something new for me, I had to get used to its syntax, its new interface. I began to study from the slides of the professor and reading the books that he advised. I slowly managed to understand the material to one degree or another. But still, I could not solve some of the harder problems from the homework.

The hardest part of this course was the midterm. It was one small game consisting of 7 subtasks. In this task, I did not succeed in the third task on the account, and since the following sub-tasks are related to the previous one, I

had problems. Thanks to Prof. Cerone for helping me to solve this problem and allowing me to complete this task at home.

After finishing the first module, we are now studying logical systems for verification of mathematical proofs in Lean. We started with propositional logic and natural deduction, which is very familiar to me, as I said from the course on Advanced Data Structures and Algorithms where we also interpreted logic by using truth values. At this stage of the course, I did not have big problems or questions. In the second section of the course regarding lambda-calculus and Curry-Howard Isomorphism, I began to experience some problems. These issues related to the lack of basic concepts and knowledge related to this section. I especially had problems writing code in Lean. I began to solve these problems by self-studying the material and asking the instructor for additional help, just as in the first module of this course.

6 Discussion

We believe that an important result of the learning process is the set of ideas sparked by the learning material. For instance, a student who is learning AI might start thinking about the implications of various applications of AI. These ideas are often silly or far fetched, but they help us motivate ourselves as students. Moreover, sharing these so-called "shower thoughts" might offer instructors a window into the minds of typical students, help them understand our thought process, and point to possible improvements to the learning material.

6.1 Alen's thoughts

As I said in the learning experience section I was skeptical about the value of formal methods. That included theorem verification tools. The reasons were twofold

1. I couldn't imagine how a mathematical proof beyond simple first-order logic examples could be expressed in a formal language. For instance, how can you express Cantor's proof of the uncountability of real numbers in Lean?

2. Why don't mathematicians use these tools?

But that changed when Prof. de Nivelle sent us a pop-science article[4] about Lean. Apparently, some mathematicians think codifying math in a formal language would add rigor to the field of mathematics. If I understood correctly, Kevin Buzzard, a mathematics professor at Imperial College London, was able to express any proof he tried in Lean. He acknowledges that it is very laborious and there are challenges to overcome, but is hopeful about the future. This article gave plausible answers to my 2 questions. First, complicated math proofs can be expressed in formal languages. After a bit of searching, I found a page with a list of proofs written in various theorem verification tools such as Coq, Lean, Mizar, etc[3]. Second, mathematicians still write proofs in English because writing proofs in a formal language is laborious. Moreover, according to Prof. Buzzard, the field of mathematics relies on authorities. Famous and respected

mathematicians dictate whether certain extremely complicated proofs are correct or not. So, legacy might also partially explain why mathematicians don't rely on formal proof verification tools.

A year before taking formal methods, I took a course on Real Analysis. I failed it spectacularly. I attribute my failure to the following reason: I lacked the intuition for solving math problems that students from math and physics departments had. I took the prerequisites, but after passing the courses I didn't practice the material, whereas math students kept on practicing. I was hoping that it would be enough to carefully follow the logic of the proofs. Unfortunately, I found that math proofs in Real Analysis were not that detailed. For me, it felt like they were skipping some steps. I assume, math students found the transitions from one step to another obvious and were able to

ll in the gaps. Now, what if Lean was used to teach Real Analysis to students like me? Perhaps, such a course wouldn't turn a CS student into a mathematician, but it will give understanding and appreciation for the foundations of mathematics. Indeed, there is an online textbook covering sets, real numbers, uncountability, and functions in Lean[5].

6.2 Ruslan's thoughts

In this section, I would like to discuss the applications of formal methods. Since the beginning of the course, I have become curious what are formal methods, why don't we use them? And I found out the following. On one of the blogs [6], I saw the following question: "What prevents the widespread adoption of formal methods?" The question was closed as biased, and most of the answers were comments like "Too expensive !!!". It is technically by answer, but explains little. Next I am going to cover the answers that I have found to my questions. They cover a broader historical picture of formal methods (FM), why they are not actually used and what people are doing to correct the situation.

Turns out there are not that many formal methods: just a few tiny groups. Different groups use the terms differently. As I understood, there are two groups of formal methods: formal specification is about the writing of precise, unambiguous specifications, and the formal verification is about the methods of proving. This includes both code and abstract systems. Not only do we use different terms for code and systems, we often use different tools to verify them. To make things even more confusing, if someone says that he is creating a formal specification, this usually means design verification. And if someone says that he is doing formal verification, this usually refers to code verification.

Proof is hard to get Proving is difficult, and it is a very nasty job. Surprisingly, formal proofs of code are often stricter than proofs written by most mathematicians! Mathematics is a creative activity but still requires rigor. Creativity goes poorly with formalism and computers. Any mathematician will immediately understand what induction is, how it works in general, and how it works in this case. But in the program for proving the theorems, everything needs to be strictly formalized. Moreover, all assumptions must be formalized, even those where most mathematicians do not bother with proof. For example, that a +

(b + c) = (a + b) + c. The program for checking theorems does not know that this is true apriori. You either have to prove it (difficult), or avoid proving it by stating it is obvious, or buy a library of theorems from someone who has already been able to prove it (expensive). Early theorem-proving programs competed in the number of built-in proof tactics and related theorem libraries.

Next, you need to get the proof itself. You can entrust this to the program or write it yourself. Usually, the problem of finding proofs automatically is not decidable. For extremely narrow cases, such as propositional logic or HM (Hindley-Milner)[7] type checking, it is "just" NP-complete. In fact, we ourselves write most of the proofs, and the computer checks its correctness. This means that you need to be well versed in: -math -computer science -the area in which you work: compilers, hardware, etc. -the nuances of the program for proving the theorems that you use, which in itself is a whole specialty

Why is this needed? It's time to step back and ask: "What's the point?" We are trying to prove that some program meets some specification. Correctness is a spectrum. There are two parts to verification: how objectively "correct" your program is and how carefully you checked the correctness. Obviously, the more verified, the better, but verification costs time and money. If we have several restrictions (performance, time to market, cost, etc.), full validation is not necessarily the best option. Then the question arises, what is the minimum check we need and what it costs. In most cases, for example, 90 % or 95 % or 99% correctness is enough for you. Maybe it's worth the time to improve the interface, and not to check the remaining 1%?

Then the question: "Is the check 90/95/99% much cheaper than 100%?" The answer is yes. It is quite comfortable to say that the code base, which we tested well and typed well, is basically correct except for a few corrections in production, and we even write more than four lines of code per day. In fact, the vast majority of malfunctions in distributed systems could have been prevented with slightly more comprehensive testing. And it's just an extension of the tests, not to mention fuzzing, property-based testing, or model testing. You can get a truly outstanding result with these simple tricks without having to get full proof.

What if typing and testing do not provide sufficient verification? It's still much easier to switch from 90% to 99% than from 99% to 100%. Cleanroom [8] is a developer practice that includes comprehensive documentation, a thorough stream analysis, and an extensive code review. Neither evidence, nor formal verification, nor even unit testing. But a properly organized Cleanroom reduces the density of defects to less than 1 bug per 1000 lines of code in production. Cleanroom programming does not slow down the pace of development and certainly goes faster than 4 lines per day. And Cleanroom itself is just one of many highly reliable software development methods that are in between the usual development and code verification. You do not need complete verification to write good software or even almost perfect. There are times when it is needed, but for most industries, it is a waste of money. However, this does not mean that formal methods are generally uneconomical. Many of the aforementioned highly

reliable methods are based on writing code specifications that you do not formally prove. In terms of verification, there are two common ways that industry benefits. First, design verification instead of code, which we will discuss later. Secondly, a partial verification of the code, which we will consider now.

Partial Code Verification For everyday tasks, it's too expensive to do a full check. How about partial? After all, you can benefit from the proof of some properties of individual code fragments. Even the simplest evidence for C programs is a great way to eliminate a huge portion of undefined behavior.

The problem is convenience. Most languages are designed either for full verification or do not support it at all. In the first case, you miss many good features from more expressive languages, and in the second case, you need a way to prove things in a language that is hostile to the concept itself. For this reason, most of the research on partial verification focuses on several high-priority languages, such as C and Java.

Design specification So far, we have only talked about code verification. However, formal methods have another side, which is more abstract and verifies the design itself, the architecture of the project. This analysis is so deep that it is synonymous with a formal specification: if someone says that he is fulfilling a formal specification, it most likely means that he defines and verifies the design.

As I've already said, proving the whole codebase is very, very difficult. But many problems in production arise not because of the code, but because of the interaction of system components. If we ignore the implementation details, for example, take it for granted that the system is capable of recognizing birds, then it will be easier for us to analyze how trees and birds, as high-level modules, fit into the overall design. On such a scale, it becomes much easier to describe things that you could not realize, such as runtime, human interactions, or the merciless stream of time. On this scale, we formalize our intentions using a high level modeling language, rather than lines of code. This is much closer to the human level, where projects and requirements can be much more ambiguous than at the code level.

For example, let's take a procedure with a rough specification "if it is called, it makes a system call to save data to the repository and processes system errors". The properties of such procedure are difficult to verify, but it is quite clear how to implement this procedure. Is the data serialized correctly? Are our guarantees violated due to incorrect input? Are we handling all possible ways of a system call failure? Now compare the high-level logging system with the specification "all messages are logged" and answer the following questions: -Are all messages recorded or only those that enter the system? -How are the messages sent? Are they delivered in the correct order? Does the channel deliver them only once? Is everything all right with the delivery? -By logging, do we mean recording forever? Is it possible to log a message and then delete it from there? Can it "hang" between recorded and unrecorded states before it is finally recorded? -What if the server explodes while recording the message? Should that be logged? -Are there any important media properties? Does the fact "media lost data" go beyond our requirements or not?

Without a formal design, it is more difficult to express the truly necessary requirements for the system. If you cannot express them, then you have no idea whether the design really meets the requirements or just seems to, but can lead to unpredictable consequences. By more formally expressing intentions and design, we can rigorously verify that we are actually developing what we need.

7 Conclusion

In this paper, we reported our background, motivation for taking this course, learning experience and we discuss some of our thoughts. The biggest difference between our learning experiences is unsurprisingly the amount of effort we put into learning the material. Alen's motivation (or lack thereof) is reflected in his approach to learning, while Ruslan puts in considerable effort into self-study. We can also note that Ruslan's discussion is more focused on the practical issues of integrating formal methods into a real-world workflow. Alen's thoughts, on the other hand, are more about his shaky math background. As can be seen in the discussion section, we are both convinced of the practical value of formal methods whether for educational purposes or in software engineering. As an ending note, learn formal methods and have fun!

References

1. Maude Homepage, http://maude.cs.illinois.edu. Last accessed 15 Nov 2019
2. Lean Website, https://leanprover.github.io. Last accessed 15 Nov 2019
3. David Barker-Plummer, Jon Barwise, and John Etchemendy.: Language, Proof, and Logic: Second Edition (2nd ed.). Center for the Study of Language and Information/SRI. (2011)
4. Number Theorist Fears All Published Math Is Wrong-VICE. fromhttps://www.vice.com/en$_us$/$article$/8xwm54/$number - therist - fears - all - published - math - is - wrong - actually?utm_campaign = sharebutton$.
5. Logic and Proof, http://leanprover.github.io. Last accessed 14 Oct 2019
6. What are the barriers that prevent widespread adoption of formal methods? - StackExchange, https://softwareengineering.stackexchange.com/questions/375342/what-are-the-barriers-that-prevent-widespread-adoption-of-formal-methods. Last accessed 14 Oct 2019
7. Hindley-Milner Interference, http://dev.stephendiehl.com/fun/006$_h$$indley_m$$ilner.html$.
8. Nascimento, Mario and Tanik, Murat: Towards Zero Defect Software: The Cleanroom Approach. (1995)