Postmortem: Teaching Haskell to Large Groups

Jurriaan Hage R. Koot
Department of Computing and Information Sciences
Utrecht University
Utrecht, The Netherlands
J.Hage@uu.nl R.Koot@uu.nl

In this paper, we describe our experiences teaching concepts of functional programming to a group of almost 300 students. In it we discuss what topics we taught, the kind of assignments we expected them to make, the operational aspects of the course, what the students thought of the course, and how they fared. We compare it to a previous incarnation of the course, and indicate where the course can still be improved. We hope our experiences can help others in a position similar to ours.

1 Introduction

In 2013, from the second week of September, until early November, we taught a first course on Functional Programming in Utrecht. At slightly under 300 students this was a course on a pretty large scale. In this paper we describe our experiences in teaching functional programming to such a large group of students. We use Haskell for teaching functional programming, since it harbours all the programming language concepts we would like to introduce in the course.

The changes we made can be summarised as follows:

- a course for teaching students new ways of programming, beyond C#
- lots of practical work for students
- the lecturer provides structure, a first encounter with the material, but it is up to the students to learn by programming
- some material was dropped in favour of other material (more details on this later)
- small assignments at first, with immediate, automatic feedback
- assignments, if fully correct, are graded further for style
- an early test to let students know where the stand
- use e-mail and domjudge system for direct contact

2 The context

Students of the Functional Programming course are second year undergraduates (unless they are retaking the course; there are in fact many such students). They have already taken courses on C# programming, databases, datastructures, and logic and sets (and at least four others, but they do not play much of a role here). The course is mandatory for computer science bachelor students, including a large number of so-called game technology bachelor students. The latter group will receive a computer science bachelor just like the ordinary computer science bachelor students, but their programme is more tailored to game design and construction. In addition to these computer science majors, there is also a smaller group of

Submitted to:
TFPIE 2014

some XX • **C** •KI (Cognitive Artificial Intelligence) students who are enrolled at the Philosophy Department. Their background includes Java • **P** •rolog programming, but typically not the other courses. More detailed numbers of the students that enrolled for the course are given later on.

The language of choice for teaching FP is Haskell, as the group at Utrecht which is responsible for the course is largely focused on that language. An extensive reader is used for the course, evolving with the developments of the language. In addition to a focus on programming in a functional language, the course is also the one that teaches students how to perform inductive proofs. In the earlier course on logic they have already seen proofs for propositional and predicate logic. The course on imperative programming is called a prerequisite, which, at our university, does not mean that students must have passed the course, but that they can expect material from that course to be necessary for understanding this one, and they cannot expect the lecturer of the current course to explain anything about the prerequisites.

A course at Utrecht University ordinarily takes place in a period of 10 weeks (the course year consists of four such periods). The final week is typically reserved for the exam. In some of the periods there is a week off somewhere in the middle where students can do a retake of a course form the previous period. Since this is the first course of the course, that is not the case here. In addition to functional programming, students are expected to take one other course in this period, and some students elect to take two • **we'll have something more on that later** •. For each course such as this, students obtain 7,5 european credits (ects), and a bachelor program amounts to 180 points. Each ects is supposedly equivalent to 28 hours of work on the part of an average student. This means that we may expect a student to spend 210 hours on average to pass the course.

The first author is the teacher responsible for this particular course, for the first time. To assist him there are nine teaching assistants, of which six are undergraduate and graduate students, and three are PhD students in the employ of the department. The second author is one of these PhD students.

Per week, there are four hours of lecture (each "hour" is 45 minutes), two on Tuesday and two on Thursday. In addition, there are four additional contact hours: two in which the students are assisted in doing their practical assignments, and two in which students are assisted at preparing themselves for the exam. These contact hours were on the same days as the lecture hours (• **details?** •). In this course, we, however, did not dictate any of this. If students wanted to, they could also have assistance on practical assignments for four hours, and do the exercises at home without assistance, for example.

When the job was given to the first author to take over FP, one of the issues with the course was that in the previous year only • **XX** • percent of the students had passed it. The department had also had received questions about the low success rate for this course during a recent visit by an external committee. Compared to other courses in the curriculum, the rate was much lower: the next lowest was about 60 percent.

Hence there was a strong motivation on the part of the lecturer to change the course so that students would be more motivated to work at it, and as a consequence more students would pass the course. There was also some hope that by changing the course, more people would be inclined to follow-up on the course by taking a bachelor course on Languages and Compilers which teaches students about regular expressions, context-free languages, and including assignments for building parsers using combinator libraries.

Incidental complications

The course was not taught under ideal circumstances. It was decided relatively late in the day that the first author would teach the course, at which time rooms were already booked for Mondays and Wednesdays. Since he is not available on Wednesday, the course had to be moved to Tuesdays and

Thursdays. However, at that time there were only few rooms left which led to many complications: not all assistance hours could be held at the same time, and some of the assistance hours were during dinner time (17-19 in the evening). The room for lecturing could only hold 210 people, quite a bit less than the registered 295. Also, the number of computers we had access to during practical hours was way less than the potential number of students.

We do know from experience that attendance trends to drop quickly after the first week, and indeed that was never much of a problem. Moving the course hours did make it necessary for us to teach some of the lectures twice, once on Wednesday (for some of the CKI students) and once on Thursday (for all others). The latter was given by the first author, the former by Jeroen Fokker. It is to be be expected that these complications will arise in the next incarnation of the course, so we shall not further discuss these issues here.

3 The course

Before we go on to discuss its actual contents, the main ideas behind the choices we made in teaching this course are the following:

- Motivate functional programming by the fact that the course teaches them valuable new programming concepts that are also useful beyond Haskell. Stress that many non-functional languages also support anonymous functions, and parametric polymorphism, and knowing these concepts makes them better programmers. In other words, we teach concepts, with Haskell as the vehicle for our teaching.
- Use lectures to motivate (and not exhaustively teach) students to work at the material.
- Programming can only be learned from doing.
- Provide for quick (or even immediate) feedback to students.
- For reasons of efficiency, automate as much as possible, in particular the grading.

The FP course does not use a book, but has its own reader that was orginally written by XX , and has since then evolved along with Haskell and the course. Since new parts are written in English, it is a mix of Dutch and English.

3.1 Reader material

The reader includes material on the following subjects:

- a general positioning of functional programming within programming
- higher-order functions
- parametric polymorphism
- functions on lists, e.g., —map—, —filter— and —foldr—, and list comprehensions
- datatypes and pattern matching
- types and type inference

These topics we expect to be part of any first time programming course in a strongly typed functional programming language. Indeed, the first test (that took place some four weeks after the course has started), included exactly these topics. And all these topics recurred in the second, final test.

After the first test, the course went on to discuss the following subjects, in the given order:

- type classes
- monads
- I/O
- inductive proofs for properties of functional programs
- embedded domain-specific languages (EDSLs)
- laziness, strictness and performance debugging

The reader has a long chapter on type classes, but spends only little time on monads. Therefore, we used additional (excellently written) material written by Graham Hutton [**?**]. For performance debugging we directed the students to Chapter 25 of the book Real World Haskell [], as well as XXX. The other topics listed above were all part of the reader.

Changes to the content and order with respect to the foregoing years

The more extensive discussion of monads, explicit strictness, and performance debugging were topics that were new to the course. We dropped finger trees from the course, and instead of discussing parser combinators (which are also discussed in the course on Languages and Compilers), we opted for a different EDSL, the financial combinators of [], and also introduced the notion of EDSL more generally.

A major change was to discuss I/O after monads had been discussed. In the previous years, I/O was discussed relatively soon, probably because it was used in a practical assignment. This year, we simply opted to have I/O only in an assignment late in the course. We could then also use I/O as another example of a monad explicitly, and we hoped this repetition would have positive effects on the students' understanding.

3.2 Exercises

The reader contains a large number of exercises, but no answers. In earlier years, answers were provided at the start of the course. This year, we decided not to supply the answers, but only told the students that we would supply at least some of them at the end of the course. (In the end, we made all of them available about a week before the exam). The motivation for this decision, is that we were under the impression that having the answers around easily led to students looking at the answers to soon, which prevented them from thinking about them first.

3.3 Assignments

In previous years, the idea was typically to have two sizable assignments (each taking 3 or 4 weeks), which were then graded by the teaching assistants. We decided to change this completely for various reasons. Most importantly, we wanted many small assignments, so that students had to deliver something almost every week. This made sure that students had to hand in something soon, which in turn allowed us to provide early feedback on how they were doing. The assignments tended to be focused on one or few new features of the language. We could in this way also ensure that there was a gradual build-up in the difficulty of the assignments. These first assignments were devised for learning the language, and the students were expected to come to their solution in a number of prescribed steps. Since we thereby had much control on how the solutions evolved, it was also possible for us to automate the part of the grading

A distinctive advantage of the small, controlled assignments was that we could then use the domjudge system [] that the second author was intimately familiar with, for providing students with immediate

feedback on their progress. This system was made to automate the grading of assignments made during programming championships, and students had already seen it in the course on Datastructures. The system gave us a number of advantages:

- On submission (students were allowed to submit any number of times), students got immediate feedback on how close they were to a completely correct solution.
- We ensured that passing all the tests, implied that the assignment fulfilled its specification completely, so during the grading phase, assistants did not have to consider whether the program was correct or not, and could concentrate on judging programming style and the like.
- Because much of the grading was automated, we could insist that students work by themselves, avoiding the issue of piggy-back riding.
- Because we could keep track of students submitting programs, we could, for example, see whether a student asking for extension had not wasted any time starting to work on the assignment.

These small controlled assignments were followed by a single, larger assignment involving the Yesod framework in which we wanted students to also experience the engineering of software of Haskell, and to allow them some more creativity. These assignments were all graded by hand. But by then the course had finished and quick feedback was not an issue anymore.

Below, we discuss in some detail what the various assignments, A0 through A5 were about:

A0 was meant only to familiarize students with the domjudge system. Since some of our students were not computer science majors, we could not assume everyone had already used it. This was our way of ensuring that start-up problems on the part of the students would not occur when it counted.

3.4 Description of the assignments

3.4.1 "Introduction"

The introductory, or zeroth, assignment was intended purely as introduction to the GHC Haskell compiler and interpreter and the DOMjudge system and consequently no points were awarded for correctly completing the assignment.

The assignment text consisted of four parts: The first part guided the students through writing a "Hello, world!" module, compiling it using the compiler, loading it into the interpreter and using the read-eval-print loop. Surprisingly, this already turned into a stumbling block for some students. While there were step-by-step instructions some familiarity with using a command line was assumed and up until this point in their studies they were able to make do with Visual Studio or Eclipse integrated development environments.

The second part showed how to do input/output in Haskell using the `interact` function, which reads a string from the standard input, applies a function to this string and writes the result back to the standard output. This function postponed the need to introduce monads early in the course. Next, the students are shown the program

```
main = interact (unlines . map reverse . lines)
```

which is used to transform the input

```
eb ot
eb ot ton ro
noitseuq eht si taht
```

into

```
to be
or not to be
that is the question
```

while introducing the concepts *higher-order function* and *function composition* on the side.

The third part asked the students to modify the programme above to output

```
to be / or not to be / that is the question
```

instead, using the `intercalate` function from the standard library, giving only a link to its documentation on the Hackage code repository as an instruction on how to use this function.

The final part instructed students on how to submit their solution to the DOMjudge system and explained how the various responses the system can give were to be interpreted.

3.4.2 "Lists"

The first real assignment introduced lists, the bread-and-butter data structure of functional languages. The students were asked to read in a database of the format

```
first last gender salary
Alice Allen female 82000
Bob Baker male 70000
Carol Clarke female 50000
Dan Davies male 45000
Eve Evans female 275000
```

and pretty print it as (note the changes in capitalization and alignment!):

```
+-----+------+------+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+------+------+
|Alice|Allen |female| 82000|
|Bob  |Baker |male  | 70000|
|Carol|Clarke|female| 50000|
|Dan  |Davies|male  | 45000|
|Eve  |Evans |female|275000|
+-----+------+------+------+
```

as well as write a selection and projection function on the rows and columns of the table.

The assignment was split into eight smaller exercises, each of which generally asked the student to implement a single function. Most exercises could be solved by composing functions from the standard library and the names of all functions required to find the model solution were given as a hint (with the explicit remark that finding a different solution not using exactly those functions was also possible).

The functions used in the model solution and given in the hints were: `all`, `elemIndex`, `filter`, `foldr`, `isDigit`, `length`, `map`, `mapMaybe`, `maximum`, `maybe`, `replicate`, `toUpper`, `transpose`, `uncurry`, `words`, `zip`, and `(!!)`.

The assignment ended with a number of open-ended reflective questions provoking students to think about the difference of programming in a functional versus an imperative paradigm.

To test the correctness of the students' solutions were designed a number carefully crafted test cases, consisting of an input database and the expected output. This approach to testing, which was taken from the algorithms and data structures courses, fails to be fine-grained enough for both the instructors and the students in a functional programming course: for the instructors it is difficult to design test cases that provide full coverage, while students are only told that they fail on a particular test case (together with the output they gave and the output that was expected) instead of which individual exercises were implemented wrongly. More advanced testing techniques are available and for the next three assignments we decided to use a QuickCheck test suite.

3.4.3 "Data structures"

The second assignment focused on using algebraic data structures and pattern matching. The goal of the assignment was to implement a noughts-and-crosses game (also known as tic-tac-toe) together with a minimax-based artificial intelligence.

The assignment is again split into smaller exercises, fourteen this time, which ask the student to implement one or two functions with a given description and type signature, possibly by defining a helper function. The exercises guided the student through defining a Rose tree data structure, modelling the game state, computing a game tree, and implementing the minimax algorithm. Compared to the previous assignment almost no hints where given in which functions from the standard library to use.

Inspired by Hughes's "Why functional programming matters", we explicitly pointed out modularity advantages of lazy functional programming to students by noting that the generation of the game tree could be decoupled from its traversal by the minimax algorithm (which in the required efficient implementation also does not require the full tree to be traversed and thus not even generated).

Like in the previous assignments the students were supplied with a starting framework that contained the type signatures of all the functions that needed to be implemented, but left their implementation `undefined`. This discouraged students from implementing the functions with a name or type signature other than specified in the assignment text and thus make sure their code compiled correctly against the test suite, and encouraged students to keep their code in a form that type checked and compiled while developing. If the students did so they were able to submit the partial solutions to the DOMjudge system and see the feedback from the test suite for the exercises that they had already completed (with the restriction that later exercises could depend on earlier ones, likely causing spurious exceptions to be raised for the later tests if the earlier exercises had not, or not completely, been implemented. Additionally, the starting framework included a game loop, allowing the students to play the game in human-versus-human, human-versus-computer, or computer-versus-computer mode once all the exercises had been implemented.

3.4.4 "Type classes"

The third assignment focussed on Haskell's type classes. The assignment was split into two parts, each consisting of a number of smaller exercises. The first part concerned containers: the student was asked to implement a `Functor` instance for the Rose tree data structure from the previous assignment; the students was introduced to the monoid algebraic structure and asked to implement sum and product `Monoid` instances for integers; finally students were asked to write a generic `fold` function operating on container-like data structure (data types with a `Functor` instance) storing elements that form a monoid. The second part involved defining an `Ord` instance (total ordering) on poker hands.

3.4.5 "Monads"

The fourth assignment focussed on monads. The assignment was again split into two parts, each consisting of a number of smaller exercises.

The first part gave a monadic interface for tossing coins and rolling dice. Students were asked to implemented a simple game based on this abstract interface. Next the student was asked to give an implementation of the monadic interface for the IO monad, using its random number generator and approximate the probability of winning the game by running it a large number of times in the IO monad. After this the student was asked to give a second implementation of the inferface for a decision tree data type (a *free monad*) and compute the exact probability of winning the game using the decision tree generate by running the game inside the free monad.

For the second part students were expected to implement an intrumented version of the state monad (discussed in Hutton's monad material) counting the number of `get` and `put` operations.

3.4.6 "Web application"

The fifth and final assignment involved developing a simple web application using the Yesod framework by pairs of students. This assignment was motivated by the desire to show how Haskell could be used in a practical situation and to make the move from problem solving and learning the language, to the engineering of applications. In particular, the use of strong static typing to prevent errors and security issues in web development. We chose the Yesod framework because it has excellent documentation for beginners in the form of the book *Developing Web Applications with Haskell and Yesod: Safety-Driven Web Development*, which is freely available online. —why it didn't work out that well—
• **friday vs. monday** •

4 Results

As usual, each course in our curriculum is followed by a student evaluation. In it students answer questions about the organization of the course, the quality of the teaching and so on.

But first, let us see how the course actually did go. • **attendance, delivery of assignments, retakes, tests (how many, how well)** •

What kind of students enroll in this course? Taken from a list of XXX students.

In the lectures we did not formally take attendance, but the first author regularly counted how many students were there. At the first lecture about 240 students attended (which meant many of them were sitting on the stairs and the ground). The attendance then gradually dropped down to about 140, but picked up again to about 170 as the more difficult topics such as type classed and monads were addressed.

Attendance of the practical hours tended to be low, particularly for the Thursday session (which was at the awkward time of 17-19) when the students were still working on the small assignments. When students started to work on the Yesod assignment attendance increased since it turned out that many students had a hard time with that one.

A nice aspect of using the domjudge system that notwithstanding low attendance at assistance, we knew that many more people were at work on the assignments. It just seems that many students did not really need the assistance. Students did not work much on (paper) exercises in class.

The first test was given on Oct. .... It included question like: compute the type of —map map—, ..., There were no multiple choice questions. In total 265 students our of the registered 295 took the exam. Grade summary: ....uniform... General conclusions: the students did really well on the type inference

question. It seems they heeded the lecturer's advice to study this well (but no explicit inquiries were made). He had promised there would be such a question in the exam.

The second, and final, test was given on Nov. .. In this case, 2.. students took the exam. The results had quite a different distribution to the first test:

It seems that this was mainly due to the fact that many of the students that scored really low on the first test had already given up. Twenty percent of the grades was determined from multiple choice questions. These were made very well. The students had the most problems with the QuickCheck question. Anecdotal feedback from students indicated that they had not expected such questions at the exam, so they had not studied it. On the other hand, they had also been told regularly that there would be an inductive proof as part of the exam, but quite a few students scored no points there. This is different from the experience we had for the first test where it turned out that many students

We provide our results by answering three questions:

Did the students do much better this year?

And were they more satisfied with the course?

Did more students opt to take Languages and Compilers?

We have seen that many more students passed this year than the previous year. But does that make them enjoy functional programming more? One indicator is a large growth in the Languages and Compilers course which is a 3rd level course on formal languages that uses Haskell as it implementation language. Although enrollment for the course grew by some .. percent, the percentage of students taking the course remains somewhat low. Of course, whether students take the course depends on many other factors as well: in the same block another mandatory 2nd year course is taught, that for most students will have a higher priority than L&C (check block).

4.1 Feedback from students

Feedback from students was received in three different ways: orally, during or after class, by e-mail, and anonymously through questionnaire.

5 Reflection

Using e-mail all the time, keeps them better aware of what is going on if they are on the mailing list. Found out later that some students were not on there, and did not bother to tell us so.

- new final assignment, comparable to the assignments of earlier years. Hand graded.
- programming style is hard to grade
- the reader? translate or drop altogether?
- plagiarism detector
- did the course become easier? Easier to pass, certainly

The financial combinators failed to strike a chord, so we'll be looking at other possibilities. Preferably, we'd like something graphical since many of our students are game/graphics oriented. This year, there was no time to work that out.

6 Conclusion

We have reported on our experiences in teaching functional programming as a mandatory course in the second year of a computer science curriculum to a large group of students.

Acknowledgements Annelies de Vries, Alexander Elyasov, Jeroen Fokker, Doaitse Swierstra