Experience Report: The Next 600 Haskell Programmers

Jasmin Christian Blanchette  Tobias Nipkow  Lars Noschinski  Dmitriy Traytel

Fakultät für Informatik, Technische Universität München, Germany

Abstract

We report on our experience teaching a Haskell-based functional programming course to over 600 students. The syllabus was organized around selected material from various sources. Throughout the term, we emphasized correctness through QuickCheck tests and proofs by induction. The submission architecture was coupled with automatic testing, giving students the possibility to correct mistakes before the deadline. To motivate the students, we complemented the weekly assignments with an informal competition.

Categories and Subject Descriptors  D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; K.3.2 [Computers and Education]: Computer and Information Science Education—Computer science education

General Terms  Algorithms, Languages, Reliability

Keywords  Haskell, functional programming, induction, testing, monads, education, competition, QuickCheck, SmallCheck

1. Introduction

This paper reports on a mandatory Haskell-based functional programming course for 618 third-semester students that we taught in the winter semester of 2012–2013 at the Technische Universität München. The course ran for 15 weeks with one 90-minute lecture and one 90-minute tutorial each week. The weekly homework was graded but those grades counted only very little toward the final grade, which was primarily determined by the final examination. To make the homework more attractive, we coupled it with an informal programming competition.

The departmental course description does not prescribe a specific functional language but focuses on functional programming concepts in general. In the previous two years, the course had been based on Standard ML. We had a strong ML background ourselves but chose Haskell because of its simple syntax, large user community, real-world appeal, variety of textbooks, and availability of QuickCheck [3]. The one feature we could well have done without was lazy evaluation; in fact, we wondered if it would get in the way.

The course was mandatory for computer science (Informatik) and information systems (Wirtschaftsinformatik) students. All had learned Java in their first semester. The computer science students had also taken courses on algorithms and data structures, discrete mathematics, and linear algebra. The information systems students had only had a basic calculus course and were taking discrete mathematics in parallel.

The dramatis personae in addition to the students where lecturer Tobias Nipkow, who designed the course, produced the slides, and gave the lectures; Master of TAs Lars Noschinski, who directed the 12 teaching assistants (TAs) and took care of the overall organization; and (Co)Masters of Competition Jasmin Blanchette (MC) and Dmitriy Traytel (CoMC), who selected competition problems and ranked the solutions.

2. Syllabus

The course covered the following topics in order. Each topic was the subject of one 90-minute lecture unless otherwise specified.

1. Introduction to functional programming [0.5 lecture]
2. Basic Haskell: Bool, QuickCheck, Integer and Int, guarded equations, recursion on numbers, Char, String, tuples
3. Lists: list comprehension, polymorphism, a glimpse of the Prelude, basic type classes (Num, Eq, Ord), pattern matching, recursion on lists (including accumulating parameters and non-primitive recursion); scopes by example [1.5 lectures]
4. Proof by structural induction on lists
5. Higher-order functions: map, filter, foldr, λ-abstractions, extensionality, currying, more Prelude [2 lectures]
6. Type classes [0.5 lecture]
7. Algebraic datatypes: data by example, the general case, Boolean formula case study, structural induction [1.5 lectures]
8. Modules: module syntax, data abstraction, correctness proofs
9. Case study: Huffman coding
10. Case study: parser combinators
11. Lazy evaluation and infinite lists
12. I/O and monads
13. Complexity and optimization
14. Advanced fun: funny functions (Ackermann, Takeuchi, etc.), proof assistants (Isabelle [9])

Most topics were presented together with examples or smaller case studies, of which we have only mentioned Boolean formulas. Moreover, two topics kept on recurring: tests (using QuickCheck) and proofs (by induction).

From day one, examples and case studies in class are accompanied by properties suitable for QuickCheck. And rather than concentrate all inductive proofs in the lecture about induction, we distributed them over the entire course and appealed to them whenever it was appropriate. A typical example: In a case study, a function is first defined via map . myg . map . myf and then optimized to map (myg . myf), justified by a proof of map (g . f) = map g . map f.

Much of the above material is uncontroversial and part of any Haskell introduction, but some choices deserve some discussion.
**Induction.** Against our expectations, induction was well understood, as the examination confirmed (Section 5). What may have helped is that we gave the students a rigid template for inductions. We went as far as requiring them to prove equations \( l = r \) not by one long chain of equalities but by two reductions \( l = t \) and \( r = t \). This avoids the strange effect of having to shift to reverse gear halfway through the proof of \( l = r \). It must be stressed that we considered only structural induction, that we generally did not expect the students to think up auxiliary lemmas themselves, and that apart from extensionality and induction all reasoning was purely equational.

In Haskell, there is the additional complication that proofs by structural induction establish the property only for finite objects. Some authors explicitly restrict the scope of their lemmas to finite lists of defined elements [14], while others prove \texttt{reverse} \((\texttt{reverse} \ xs) = xs\) without mentioning that it does not hold for partial or infinite lists [6]. Finite partial and infinite lists complicate the picture. Although some authors discuss them [5, 14], we avoided them in our course—undediﬁned alone is a can of worms that we did not want to open.

**Abstraction functions.** In the lecture on modules and data abstraction, we also showed how to prove correctness of data representations (e.g., the representation of sets by lists). This requires an abstraction function from the representation back to the abstract type that must commute with all operations on the type. As the corresponding homework showed, we failed to convey this. In retrospect, it is outside the core functional programming syllabus, which is why it is absent from all the textbooks.

**Input/output.** I/O was covered toward the end of the course because it is connected with the advanced topic of monads. In retrospect, we could have covered I/O earlier without mentioning monads. This would have helped to dispel the feeling among some students that Haskell is just a glorified pocket calculator.

**Laziness.** Haskell’s lazy evaluation strategy and infinite objects played only a very minor role and were introduced only toward the end. Initially, we were worried that laziness might confuse students when they accidentally stumble across it before it has been introduced, but this was not reported as a problem by any of the TAs. But it meant that we could not give the students a good operational model of the language: All they knew initially was that equations were applied in some unspecified order. Even after we had explained laziness, it remained unclear to many students how exactly to determine what needs to be evaluated.

**Complexity and optimization.** Complexity considerations are seriously complicated by laziness. We found that the book by Bird [1] offered the best explanation. For time complexity, he notes that assuming eager evaluation is easier and still gives an upper bound. Therefore, we simply replaced lazy by eager evaluation for this lecture. The principles then applied to most programming languages, and one can cover key optimizations such as tail recursion.

3. **Exercises**

Each week we released an exercise sheet with group and homework assignments. The main objective of the homework was to have the students actually program in Haskell. The submission infrastructure periodically ran automatic tests, giving the students fast feedback and an opportunity to correct mistakes before the deadline.

3.1 **Assignments**

A typical assignment sheet contained between three and five group exercises and about as many homework exercises. The group exercises were solved in 90-minute tutorial groups. There were 25 such groups, each with up to 24 students. Each exercise focused on a specific concept from the week’s lecture. Many were programming exercises, but some required the students to write QuickCheck tests, evaluate expressions, carry out proofs, or infer an expression’s type.

The homework assignments, to be solved individually, covered the same topics in more depth, sometimes in combination. They were optional, but the students who collected at least 40% of the possible points were awarded a bonus of 0.3 to the final grade, on a scale from 1.0 (\( \approx A^+ \)) to 5.0 (\( \approx F \)). The reason for this system is fear of plagiarism. In the end, 281 students claimed the bonus. Furthermore, the (Co)MCs nominated one of the assignments to count as part of the competition (Section 4).

Overall, the assignments were fairly well understood, perhaps because they closely followed the lectures. There were a few important exceptions.

A week 6 group problem consisted of registering the polymorphic function \( a \rightarrow b \) as an instance of the \texttt{Num} type class, so that \( (f + g) x \approx f x + g x \) and similarly for the other operators. Many students did not understand what their task was, or why one would register functions as numbers; and even those who understood the question had to realize that \texttt{b} must be an instance of \texttt{Num} and fight the problem’s higher-order nature. We had more success two weeks later when we redid the exercise for a \texttt{Fraction} datatype and gently explained why it makes sense to view fractions as numbers. (Datatypes had been introduced in the meantime.)

Less surprisingly, many students had issues with \( \lambda \)-abstractions. They tended to use \( \lambda \texttt{s} \) correctly with \texttt{map} and \texttt{filter} (although many preferred list comprehensions when given the choice), but other exercises revealed the limits of their understanding. One exercise required implementing a function \texttt{fixpoint} \( eq \ f \ x \) that repeatedly applied \( f \) to \( x \) until \( f^{n+1} x \equiv eq \ f^n x \) and then using this function to solve concrete problems. Another exercise featured a deterministic finite automaton represented as a tuple, where the \texttt{δ} component is represented by a Haskell function. Finally, the parser combinator exercise was almost universally boycotted.

One difficulty we continually faced when designing exercises is that the Internet provides too many answers. This was an issue especially in the first few weeks, when little syntax has been introduced. We did our best to come up with fresh ideas and, failing that, obfuscated some old ideas.

3.2 **Submission and Testing Infrastructure**

The university provides a central system for managing student submissions, but we built our own infrastructure so that we could couple it with automatic testing. Our submission system combines standard Unix tools and custom scripts. The students were given a secure shell (\texttt{ssh}) account on the submission server. They had to upload their submissions following a simple naming convention. The system generated test reports every 15 minutes using QuickCheck and sent them by email. Many students appear to have improved their submissions iteratively based on the system’s feedback. The final reports were made available to the TAs but had no direct effect on grading.

To increase the likelihood that the submissions compile with the testing system, we provided a correctly named template file for each assignment, including the necessary module declarations and stub definitions \( f = \texttt{undefined} \) for the functions to implement. Nonetheless, many students had problems with the naming scheme (there are surprisingly many ways to spell “exercise”), causing their submissions to be ignored. These problems went away after we started providing a per-student web page listing the status of all their assignments and announced a stricter grading policy.

A few exercises required writing QuickCheck properties for a function described textually. These properties had to take the function under test as argument, so that we could check them against secret reference implementations. Since higher-order arguments had
not yet been introduced, we disguised the argument type using a
type synonym and put the boilerplate in the template file.

The test reports included the compilation status, the result of
each test, and enough information about the failed tests to identify
the errors. The tests themselves were not revealed, since they often
contained hints for a correct implementation. In cases where the
input of the test case did not coincide with the input of the tested
function, we had to explain this in the description or provide more
details using QuickCheck’s `printTestCase` function. Some care
was needed because the function under test can throw exceptions,
which are not caught by QuickCheck because of the lack evaluation
of `printTestCase`’s argument. We used the `Control.Spoon`
package to suppress these exceptions.

To make the output more informative, we introduced an operator
`==?` that compares the expected and actual results and reports
mismatches using `printTestCase`.

We did not find any fully satisfactory way to handle very slow
and nonterminating functions. QuickCheck’s `within` combinator
fails if a single test iteration takes too long, but these failures are
confusing for correct code. Instead, we limited the test process’s
runtime, possibly leaving students with a truncated report.

3.3 Test Design

As regular users of the Isabelle proof assistant, we had a lot of
experience with Isabelle’s version of QuickCheck [2]. The tool is
run automatically on each conjectured lemma as it is entered by the
user to exhibit flaws, either in the lemma itself or in the underlying
specification (generally a functional–logic program). Typically, the
lemmas arise naturally as part of the formalization effort and are
not designed to reveal bugs in the specification.

We designed our Haskell tests to expose the most likely bugs
capture the main properties of the function under test. We
usually also included a test against a reference implementation. We
soon found out that many bugs escaped the test suite because the
Haskell QuickCheck’s default setup is much less exhaustive than its
Isabelle namesake’s. For example, the Haskell random generator
tends to produce much larger integers than the Isabelle one; as a
result, random lists of integers rarely contain duplicates, which are
essential to test some classes of functions. Worse, for polymorphic
functions we did not realize immediately that type variables are
instantiated with the unit type () by default (a peculiar choice
to say the least). In contrast, Isabelle’s version of QuickCheck
supports random testing, exhaustive testing (cf. `SmallCheck` [11]),
and narrowing (cf. Lazy `SmallCheck` [11], Agsy [8]), the default
number of iterations is 250, and type variables are instantiated by
small types. The differences between the two QuickCheck versions
became painfully obvious with the competition exercise of week 5,
as we will see in Section 4.

Following these initial difficulties, the Master of TAs was ap-
pointed Master of Tests and put in charge of setting up the test-
ing framework properly. He immediately increased QuickCheck’s
number of iterations and decreased the maximum size parameter.
He regained control by defining custom generators and instantiating
type variables with small types. He also started using `SmallCheck`
to reliably catch bugs exposed by small counterexamples.

3.4 Plagiarism Detection

We considered it important to detect and deter plagiarism, both
because individual bonuses should be earned individually and be-
cause learning functional programming requires doing some pro-
gramming on one’s own. Our policy was clear: Plagiarism led to
forfeiture of the bonus for all involved parties.

To identify plagiarists, we used Moss [12] extended with a
custom shell script to visualize the results with Graphviz [4]. The
resulting graph connects pairs of submissions with similar features,
with thicker edges for stronger similarities. Figure 1 shows an
anonymized excerpt of the output for week 3.

A noteworthy instance of unintended sharing is the complete
subgraph of thick edges in the middle of Figure 1. One of the
involved students has used Pastebin (http://pastebin.com/) for his own purposes, without realizing that it would be indexed by
Google and picked up by other students. (He received a warning.)
Moss’s results are imprecise, with many false positives, so they
must be analyzed carefully. Functional programming often allows
short, canonical solutions. Unusual naming conventions, spacing,
or bugs are useful clues. One could have thought that the recent
German plagiarism scandals, which eventually cost two federal
ministers their Dr. title and minister position, would have cured the
country for some time. Sadly, we had to disqualify 29 students.

4. Competition

Our main inspiration for the programming competition has been
CADE’s Automated Theorem Prover System Competition (CASC)
[13], organized by Geoff Sutcliffe since 1996. We have been enter-
ing Isabelle since 2009 and have noticed CASC’s motivating im-
pact on the theorem proving community. We were also moved by
our late colleague Piotr Rudnicki’s strong opinion in favor of con-
tests in an educational context [10]:

I am dismayed by the watering down of the curriculum
at CS departments which does not push the students to
their intellectual limits. This wastes a lot of talented people
who, under these conditions, have no chance to discover how
talented and capable they really are. The programming
contests attract a substantial fraction of the most talented
students that we have; I enjoy working with them and they
seem to enjoy doing it too.

The Heavenly Father, with his unique sense of humor,
has distributed the mental talents in a rather unpredictable
way. It is our role to discover these talents and make them
shine. If we do not do it, then we—the educators—will end
up in Hell. And I would rather not get there just for this
one reason.

For our own contest, each week we selected one of the pro-
gramming assignments as a competition problem. We also fixed a
criterion for ranking the correct entries. By enclosing their solu-
tions within special tags, students became competitors. Each week,
rank \( i \in \{1, \ldots, 20\} \) brought in \( 21 - i \) points. The five students cu-
mulating the most points were promised “tasteful” trophies.

Once the entries had been tested and ranked, we published the
names of the top 20 students on the competition’s web page ¹
and updated the cumulative top 20. To avoid legal issues regarding
privacy, we inserted a notice in the assignment sheets, making it
clear that the competition is an opt-in. The list of winners was

1 http://www21.in.tum.de/teaching/inf02/KS1213/
wettbewerb.html
followed by a discussion of the most remarkable solutions, written by the MC or the CoMC, in self-important third-person style, something that appears to have struck a chord with the students.

An unexpected side effect of the competition is that it provided a channel to introduce more advanced concepts, such as higher-order functions, before they were seen in class. The criteria were designed to raise the students’ awareness of engineering trade-offs, including performance and scalability, even though these topics were beyond the scope of the course.

As is to be expected, participation went down as the session progressed. We tended to lose those students who were not in the cumulative top 20, which indicates that we should have made it a top 50. The optional exercises attracted only the hard core. We have the testimony of a student who, after gathering enough points to secure the grade bonus, skipped the mandatory exercises to focus on the grade bonus. Thus, our experience corroborates Rudnicki’s: Contests motivate talented students, who otherwise might not get the stimuli they need to perform.

The (Co)MCs revealed the last competition week’s results in an award ceremony during the last lecture, handing out the trophies and presenting characteristic code snippets from each winner.

Because of the various ranking criteria, and also because students knew that their solutions could turn up on the web page, the competition triggered much more diversity than usual assignments. Looking at the solutions and ranking them was fascinating. Each exercise by Jacques Haguel. By having to keep the token count low, students were encouraged to focus on the general case.

The winner’s solution had 13 tokens (excluding the left-hand side and counting ‘max’ as one):

\[
\text{max } x \ y \ z = \min \ x \ y \ \text{max} \ z = \max \ z \ 2
\]

Here the concision was attained at the expense of simplicity, to the save on parentheses (\(f \text{max}\)) side and counting exercise by Jacques Haguel. By having to keep the token count low, arguments.

Task: Write a function that adds the squares of the two largest of its

A few competitors exploited the explicit application operator \(\_\) to save on parentheses (\(f \_ g \ x \_ f \_ g \ x\)). Using only syntaxes and functions seen in class, a 25-token solution was possible:

\[
x \_ x \_ y \_ y \_ z \_ z \_ a \_ a
\]

where \(a = \min \ x \ (\min \ y \ z)\)

The median solution had a 3-way case distinction. There were plenty of 6-way distinctions, and one entry even featured a correct 10-way distinction using \(<\) and \(\geq\), complete with 64 needless parentheses, totaling 248 tokens. This provided the ideal context for the MC to quote Donald Knuth [7, p. 56] on the competition’s web site: “The ability to handle lots of cases is Computer Science’s strength and weakness. We are good at dealing with such complexity, but we sometimes don’t try for unity when there is unity.”

To count tokens, we initially used a fast-and-frugal Perl script. However, many students asked us to make the program available, so we replaced it by a bullet-proof Haskell solution based on Niklas Broberg’s lexical analyzer (Language.Haskell.Exts.Lexer).

### Week 1: Sum of Two Maxima’s Squares (434 Entrants)

**Task:** Write a function that adds the squares of the two largest of its arguments \(x, y, z\). **Criterion:** Token count (lower is better).

The task and criterion were modeled after a similar Scheme exercise by Jacques Haguel. By having to keep the token count low, students were encouraged to focus on the general case.

The winner’s solution had 13 tokens (excluding the left-hand side and counting ‘max’ as one):

\[
\text{max } x \ y \ z = \min \ x \ y \ \text{max} \ z = \max \ z \ 2
\]

Here the concision was attained at the expense of simplicity, to the point that we felt the need to verify the solution with Isabelle. Lists appeared in several of the top 20 solutions:

\[
\text{sum } \_ \text{tail } \_ (\_2) \ 'map' \ 'sort' \ [x, y, z] \\
\text{sum } [x \_ x | x \_ \text{tail } (\text{sort } [x, y, z])]
\]

A few competitors exploited the explicit application operator \(\_\) to save on parentheses (\(f \_ g \ x \_ f \_ g \ x\)). Using only syntaxes and functions seen in class, a 25-token solution was possible:

\[
x \_ x \_ y \_ y \_ z \_ z \_ a \_ a
\]

where \(a = \min \ x \ (\min \ y \ z)\)

The median solution had a 3-way case distinction. There were plenty of 6-way distinctions, and one entry even featured a correct 10-way distinction using \(<\) and \(\geq\), complete with 64 needless parentheses, totaling 248 tokens. This provided the ideal context for the MC to quote Donald Knuth [7, p. 56] on the competition’s web site: “The ability to handle lots of cases is Computer Science’s strength and weakness. We are good at dealing with such complexity, but we sometimes don’t try for unity when there is unity.”

To count tokens, we initially used a fast-and-frugal Perl script. However, many students asked us to make the program available, so we replaced it by a bullet-proof Haskell solution based on Niklas Broberg’s lexical analyzer (Language.Haskell.Exts.Lexer).

**Week 2: Duplicate-free Reversed Permutations (338 Entrants)**

**Task:** Write a function \(\text{perms}\) that returns the list of all permutations of its input word, without duplicates and in reverse lexicographic order. **Criterion:** Token count; speed (for breaking ties).

The problem statement suggested to use the standard functions \(\text{delete, nub}\) (which removes duplicates), \(\text{reverse, and sort}\). The median solution was

\[
\text{perms } [] = ["] \\
\text{perms } x : s = \text{reverse } (\text{sort } (\text{nub } [[x] \_ p | x \_ xs, p \_ \text{perms } (\text{delete } x \ x s)]))
\]

It is exponentially slow on inputs of the form \(a^n\), even though the result is the singleton \([a^n]\). To improve performance, it is enough to apply \(\text{nub}\) directly on the input word \(xs\) when selecting \(x\), rather than on the entire list comprehension. The same can be done with \(\text{reverse}\) and \(\text{sort}\), something only two students thought of.

### Week 4: Risers and Fallers (274 Entrants)

The lecture presented a function that lists all maximal ascending sublists (“risers”) in a list over a totally ordered type \((\text{Ord a})\)–e.g., \(\text{risers } [1, \ 2, \ 4, \ 3, \ 1, \ 2] = \{[1, \ 2, \ 4], \ [3, \ 1, \ 2]\}\).

**Task:** Write a variant of \(\text{risers}\) that alternates between ascending and descending sublists–e.g., \(\text{upAndDowns } [1, \ 2, \ 4, \ 3, \ 1, \ 2] = \{[1, \ 2, \ 4], \ [3, \ 1, \ 2]\}\). **Criterion:** Token count.

A few competitors took “descending” to mean \(>\) instead of \(>\). Since the statement was not entirely clear and this deviation often escaped our QuickCheck test suite, we tolerated it. The shortest solution featured this alternative worldview:

\[
\text{map } \text{fst } ('\map') \ '\text{groupBy' } (==) \ 'on' \ 'snd' \\
\text{(zip } x : s \_ True \ : \ ('\text{zipWith' } (==) \ xs \_ \text{tail } xs)\)
\]

The same student found an even shorter solution that reused a function \(\text{irregularChunks' : [Int]} \_ \to \ [a] \_ \to \ [[a]]\) from the same exercise sheet:

\[
\text{map } \text{length } (\text{group } True \ : \ ('\text{zipWith' } (==) \ xs \_ \text{tail } xs)) \ '\text{irregularChunks' } x s
\]

Although the instructions were clear about the need to provide a complete solution, “the MC and his acolytes [were] nonetheless extremely pleased to see so much creativity at work.”

There was a lot of variation among those solutions that adhered to the standard worldview. Some students extended the \(\text{risers}\) implementation with an additional argument that is initially \(isiert\) and is \(f\text{ip'}\)d in each recursive call. A variant of this used a pair of arguments \(<=\), \(>=\) that are swapped in the recursive call. One entry started with the infinite list \(\text{cycle } [[<>, \ (>=)]\), using the head to compare elements and passing the tail in the recursive call.

### Week 5: Quasi-subsequences (206 Entrants)

**Task:** Write a function that tests whether a list \([x_1, \ldots, x_n]\) is a quasi-subsequence of a list \(ys\), meaning that it is either a subsequence of \(ys\) or that there exists an index \(k\) such that \([x_1, \ldots, x_{k-1}, x_k, \ldots, x_n] = \text{a subsequence of } ys\). **Criterion:** Speed.

Thomas Genet shared this example with us. The problem statement mentioned that the MC’s solution took 0.4 s for \(\text{quasiSubseq } [1 \ldots N] \_ \to \ [N] \_ \to \ [2 \ldots N - 1] \_ \to \ [1]\) with \(N = 50,000\). To rank the solutions, we ran some additional QuickCheck tests, filtering out 43 incorrect solutions from the 143 that compiled and passed all the official tests. Then we tried various examples, including the one above with different \(Ns\), and eliminated solutions that reached the generous timeout. Some examples had a huge \(xs\) and a short \(ys\). This produced 20 winners, whom we listed on the web site. The algorithms varied greatly and were difficult to understand. One of the TAs, Manuel Eberl, contributed an automaton-based solution. The MC’s solution had a dynamic programming flavor.
The story does not end here. Having noticed more bugs in the process, we speculated that some of the top 20 entries might be incorrect. Prompted to action by Meta-Master Nipkow, we rechecked all 20 solutions using Isabelle’s implementation of QuickCheck and found flaws in 6 of them. We did not penalize their authors but took a second look at Haskell’s testing capabilities (cf. Section 3.2).

**Week 6: Email Address Anonymizer (163 Entrants)**

**Task:** Write a function that replaces all email addresses in a text by an anonymized version (e.g., \texttt{p...q...df......c...}). **Criterion:** Closeness to the official definition of email address.

The task idea came from Koen Claessen. The statement suggested a simple definition of email addresses, which is what most students implemented, but pointed to RFCs 5321 and 5322 for a more precise definition. Our goal was to see how students react to unclear requirements. Of course, the RFCs partly contradicted each other, and it was not clear whether the domains had to be validated against the rules specified in earlier RFCs. It was also left to the student’s imagination how to locate email addresses in the text.

This task was, by far, the most despised by the students. It was also the most difficult to rank to be fair to those who invested many hours in it but failed some simple test we had design to rank the solutions. We revised the ranks upward to comfort the more vocal participants, turning the top 20 into a top 45.

**Weeks 8–9: Boolean Solver (Optional, 14 Entrants)**

**Task:** Write a Boolean satisfiability (SAT) solver. **Criterion:** Speed.

To avoid repeating the week 6 debacle, we suggested five optimizations to a DPLL algorithm that would be evaluated in turn:

1. Eliminate pure positive variables.
2. Select short clauses before long ones.
3. Select frequent literals before infrequent ones.
4. Use a dedicated algorithm for Horn formulas.
5. Use a dedicated algorithm for 2CNF.

Obvious variants of these optimizations would be invoked to break ties. The Meta-Master promised a Ph.D. degree for polynomial solutions (to no avail).

For the evaluation, we needed to devise problems that can be solved fast if and only if the heuristic is implemented. Showing the absence of an optimization turned out to be much more difficult than we had anticipated, because the various optimizations interacted in complicated ways, and the exact mixture varied from solution to solution. To make matters worse, often the optimizations were implemented in a naive way that slowed down the solver (e.g., reprocessing the entire problem each time a literal is selected to detect whether an optimization has become applicable).

Unlike for previous weeks, this problem gave no homework points. In exchange, it was worth double (40 points), and the students had two weeks to complete it. Also, we did not provide any QuickCheck tests, leaving it to the students to think up their own (“just like in real life”). There were 14 submissions to rank, as well two noncompetitive entries by TA Eberl and one by the (Co)MCs. We found bugs in 8 of the submissions, but gave these incorrect solutions some consolation points to help populate the week’s “top 20.” The two best solutions implemented optimizations 1 to 4 and pure negative literal elimination, but neither 2CNF nor dual-Horn.

**Week 12: Word Enumeration with Censorship (98 Entrants)**

**Task:** Write a function that systematically enumerates all words over a given finite alphabet, excluding words containing given forbidden words (f-words) as subwords. **Criterion:** Speed.

We tested various combinations of input sizes, including highly redundant inputs. The best solutions compiled the f-words into an efficient, redundancy-free data structure; for example, if the word \texttt{a} is forbidden, it should not matter whether \texttt{aa} is also forbidden. Some inputs lead to finite outputs—e.g., the alphabet \texttt{[a]} and the f-word set \texttt{[aaa]} produce \texttt{[\epsilon, a, aa]}; the crème de la crème would detect this condition and stop instead of spinning forever.

**Week 13: Programmatic Art (Optional, 10 Entrants)**

**Task:** Write a program that generates a pixel or vector graphic in some standard format. **Criteria:** Aesthetics and technique.

There were no constraints concerning the subject of the picture or its generation. While the impression had annoyed the students in the email anonymizer assignment, here it was perceived as a most welcome artistic freedom. The creations’ visual nature was a perfect match for the award ceremony, where the weekly and final results were presented.

The students were asked to turn in both the program and a generated picture. The Meta-Master and the Master of TAs rated the aesthetics on a scale from 1 to 10. The remaining 10 points were issued by the (Co)MCs for “technique,” mostly as a safeguard against cheaters.

Five of the ten solutions were fractal generators. Two students “misunderstood” the exercise: one handed in a generated ASCII-art text file, another used Network.Curl.Download to retrieve Vincent van Gogh’s “Starry Night” from Google. The latter secured the top score for aesthetics but was punished with a 2 for technique.

The winner—a student called Julius who had earned the nicknames Caesar, Napoleon, Teen Queen, Vladimir, and Charlemagne over the course of the term—had been visibly inspired by Piet Mondrian’s famous “Compositions” (Figure 2). His randomized solution could generate arbitrarily many fake Mondrians. Incidentally, his personality was a congenial match for the MC’s, as he used the royal we and signed his emails Julius Caesar.

**Examination**

The final grades were based on a two-hour examination. Students were allowed to bring one hand-written “cheat sheet.” They needed 20 of 50 points to pass. The results were then translated to a 1.0 to 5.0 scale. The 0.3 homework bonus was awarded only to those who had passed the examination.

The problems were similar to the group exercises but avoided more advanced or mundane topics (e.g., modules and data abstraction). The examination was designed so that the best students could finish in one hour. There were nine problems to solve:

1. Infer the types of given expressions.
2. Implement the same simple function using recursion, using a list comprehension, and using higher-order functions (e.g., map, filter).
3. Implement a function that lifts a variable renaming function to a logical formula datatype.
4. Put given function definitions in equivalence classes.
5. Prove \texttt{map f (concat xxx) = concat (map (map f) xxx)}.
6. Choose two test functions from a given set that together constitute a complete test suite for a function.
7. Evaluate given expressions step by step.
8. Implement \( n! \) tail-recursively.

9. Write an I/O program that reads the user’s input line by line and prints the total number of vowels seen so far.

Perhaps because we had no previous experience in teaching Haskell, the marking revealed many surprises. Our impressions are summarized below for each problem.

1. Many students who intuitively understood types and type inference in practice had problems applying their knowledge in a more abstract context. They often forgot to instantiate type variables or to remove the argument type when applying a function. For example, \( \text{filter} \) was often typed as \([\text{a}] \rightarrow [\text{a}] \) or even \([\text{a} \rightarrow \text{Bool}] \rightarrow [\text{a}] \rightarrow [\text{a}] \). Tellingly, one of the best students lost 2.5 of 5 points here, while answering all the other questions correctly.

2. The definitions based on a list comprehension (e.g., \( f \{x \leq \alpha \} x \rightarrow \{x \leq \alpha \} \) were usually correct. The corresponding map\(-\)filter version proved more challenging. The recursive definitions were mostly correct but sometimes lacked a case.

3. The formula datatype featured both a directly recursive constructor (for logical negation) and a pair of recursive constructors through lists (for \( n \)-ary conjunctions and disjunctions). The recursion through the list, using map (rename f), confused many (although it had been covered in several exercises). Some solutions managed to change the shape of the formula, rewriting \( n \)-ary expressions into nested binary expressions. The pattern matching syntax was also not universally understood, and the constructors were often missing in the right-hand sides.

4. The problem presented four similar function definitions, including one that was the pointfree version of another one, one that had the two arguments permuted, and one that had a needless argument. Most of the answers correctly related the pointful and pointfree versions of the function, but the justifications often left much to be desired.

5. The proof by induction posed little problems. Presumably the students had the induction template on their cheat sheet. Quite a few followed the template too slavishly, claiming to be doing an induction on \( x \) instead of \( \alpha \). Another common mistake was to take \( \alpha \{\} \) as the base case.

6. The function to test operated on lists. There were seven tests to choose from: tests for the \( \{\} \), \( [x] \), and \( : x \{\} \) cases, a distributivity law, a length law, and two properties about the content of the result list. Obvious solutions were \( \{\} \) with \( x : \alpha \) or \( \{\} \) with distributivity, but there were many other combinations, most of which we discovered while marking. For example, the length law implies the \( \{\} \) case, and the \( \{\} \) and \( x : x \{\} \) cases, taken together, imply the \( \{\} \) case.

7. The order of evaluation was not understood by all. We were downright shocked by some of the answers provided for \( (x \rightarrow (y \rightarrow (1 + 2 + x))) \). We were prepared to see solutions where the two arguments are swapped, but not monstrosities such as \( (y \rightarrow (V \rightarrow (1 + 2 + 4))) \) as the end result.

8. Many students attempted to dodge tail recursion by using library functions such as foldl or product. The problem statement could have been clearer.

9. The monadic solutions were surprisingly good, perhaps due to the students’ familiarity with imperative programming. The main difficulty was to keep track of the cumulative vowel count. Many solutions simply printed the count of each line instead. Another common mistake was to use the monadic syntax \( \langle \) instead of let to bind nonmonadic values.

Some statistics: 552 students registered for the exams. 432 students took the first examination. 334 passed it. 39 secured the top grade (1.0), with at least 47 points. Five had a perfect score. A repeat examination is scheduled for those who failed the first time.

6. Conclusion

Teaching functional programming using Haskell has been an enjoyable experience overall. As is usually the case for functional programming, the feedback from students was mixed. If we have failed to convince some of them of the value of functional programming, we have also received many testimonies of students who have “seen the light,” and some of the serious competitors told us the course had been the most fun so far.

For future years, we plan to either leave out some of the more advanced material or enhance its presentation. Type inference is one topic we downplayed this year; it should be possible to present it more rigorously without needing inference trees. On the infrastructure side, we want to develop tool support for simple proofs by induction, in the form of a lightweight proof assistant.

Acknowledgments

We thank all the people involved in giving the course, including the TAs and our colleagues from the Chair for Logic and Verification. We also thank Koen Claessen and Philip Wadler for sharing their slides with us and Mark Summerfield for suggesting textual improvements to a draft of this paper.

References


