Teaching Simple Constructive Proofs with Haskell Programs

Matthew Farrugia-Roberts
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
matt.farrugia@unimelb.edu.au

Harald Søndergaard
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
harald@unimelb.edu.au

We teach a large-enrolment university course in logic, discrete structures, formal languages, and computability. Over the last few semesters we have explored using Haskell alongside a traditional mathematical formalism to offer our computer science students a programming perspective on often complex mathematical topics. In this extended abstract, we present our approach to partially digitising certain proof-style exercises involving simple constructive arguments as Haskell programming exercises. We give examples of digitised proof exercises in logic and formal languages. A full paper will include more examples, a discussion of limitations, and a catalogue of real student submissions.

1 Introduction

We teach *Models of Computation*, a large-enrolment university course on introductory topics in logic, discrete mathematics, formal languages, and computability [6]. Most of our students take the course as part of their degree in computer science or software engineering. Elsewhere in their courses, these students learn how to use code to build complex software systems. In our course, an important goal is for our students to learn the language of mathematical modelling and proof, to help think and communicate with precision and rigour.

Over several years, we have experimented with a 'programming approach' to learning and assessment activities, via the Haskell programming language and a web-based programming platform. We use Haskell as a stepping stone to traditional mathematical formalism, and a *lingua franca* for our students to express themselves in exercises spanning a rather broad syllabus. We aim to (1) digitise elements of our course for the logistic and pedagogical benefits, and (2) offer our students a programming perspective on our mathematically demanding syllabus, which they might more readily assimilate given their computer science background. We have discussed this approach in recent work with Bryn Jeffries [2].

We have found Haskell a flexible language for exercises in myriad topics, from logic to formal languages and potentially beyond [2]. However, traditional exercises of one class—written proofs—have been generally difficult to digitise. Nevertheless, we have designed Haskell exercises to partially capture certain simple constructive proofs. In particular, our students implement the central *construction algorithm* of the proof as a Haskell function. By offering on-demand compiler feedback and automated testing, we are able to use these exercises as a supplement to a small remainder of pen-and-paper proofs.

This paper presents our approach to partially digitising constructive proof exercises, as part of our broader initiative in digitising our curriculum with Haskell. We begin by describing our course, our students, and our programming approach (Section 2). We then sketch an analogy between constructive proofs and Haskell functions (Section 3), and explore several examples of constructive proof exercises in logic and formal languages (Section 4). We acknowledge several similar initiatives in Section 5.

In a full version of this paper, we plan to discuss additional examples. We also plan to address pedagogical limitations of the analogy between proofs and programs (Appendix A), and to catalogue a sample of student submissions to identify common correct and incorrect answers (Appendix B).

Submitted to:
TFPiE 2022

© M. Farrugia-Roberts & H. Søndergaard
This work is licensed under the
Creative Commons Attribution License.

<table>
<tr><th>Week</th><th>Topic in lectures and tutorials</th><th colspan="2">Haskell exercises for learning and assessment</th></tr>
<tr><td>1</td><td>Introduction</td><td rowspan="4">Introduction to basic Haskell (self-paced, self-contained tutorial on functions, recursion, lists, basic algebraic data types)</td><td rowspan="3"></td></tr>
<tr><td>2</td><td rowspan="4">Logic (syntax and semantics of propositional and predicate logic, and mechanised reasoning via resolution algorithms)</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td rowspan="8">Worksheets: 6% (four fortnightly formative tasks on algorithms for propositional logic, regular languages, and formal grammars)</td></tr>
<tr><td>5</td><td rowspan="3">Assignment 1: 12% (mathematical and algorithmic challenges in logic)</td></tr>
<tr><td>6</td><td rowspan="2">Discrete mathematics (sets, functions, relations, termination)</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td rowspan="4">Formal languages (finite automata, regular expressions, context-free grammars, pushdown automata, Turing machines)</td><td rowspan="4">Assignment 2: 12% (mathematical and algorithmic challenges in discrete mathematics and formal languages)</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td><td>Computability (undecidability)</td><td></td><td></td></tr>
<tr><td>Exams</td><td></td><td>Exam: 70% (3 hours, all topics)</td><td></td></tr>
</table>

Table 1: *COMP30026 Models of Computation*, example semester calendar.

2 Background

Our course is an introduction to logic, discrete mathematics, formal languages, and computation. Table 1 gives an overview of the topics studied. The emphasis of the intended learning outcomes is in (1) *applying* topics in logic and discrete mathematics to reason about computational problems, and (2) *analysing* and *creating* computational models (from finite-state automata to Turing machines) [6].

In particular, we consider it a learning goal for the students to be able to apply their understanding of first-order logic and to analyse computational models by carrying out simple proofs regarding these models. For example, a student should be able to prove simple properties of formal language classes, show by construction that a given language belongs to a given class, or prove that a given language is outside a class by applying a pumping lemma or a reduction argument. Though the emphasis is on applications in computation we also study proofs in other areas such as propositional and predicate logic.

The course has a large enrolment—well over 500 students. Most students take the course for either their undergraduate major or their coursework graduate program, in either computer science or software engineering. Our students are familiar with one or more programming languages as a prerequisite. Haskell is *not* a prerequisite, though some students take a concurrent elective in functional programming. Most have some university-level mathematics experience, but the course is considered mathematically demanding by many of our students—especially when it comes to learning to write proofs.

We have adopted Haskell exercises in learning and assessment tasks to offer our computer science students a programming perspective on the mathematical topics in our syllabus. The exercises are delivered through a state-of-the-art web-based programming platform, Grok Academy (Grok) [3], allowing us to capture additional benefits by automating rapid feedback and grading. As of 2021, we retain a small number of traditional pen-and-paper exercises in assignments, primarily for exercising and assessing students' proof-writing skills. Haskell exercises comprise all other assessment, including the final exam. We detail this approach in recent work with Bryn Jeffries of Grok [2].

Since Haskell is not a prerequisite, our semester begins with a self-paced introduction to basic Haskell. Grok is an ideal platform for this introduction. Designed for learning to program, Grok is already familiar to many of our students who have previously used it to learn Python, C, and/or Java.

However, programming is not one of our direct learning goals. Our students' main use of Grok is not for learning Haskell, but for learning *with* Haskell. We offer much formative and summative assessment in the form of Haskell programming exercises. Notably we use Haskell as a *flexible digital answer format*. This includes not only implementation tasks (where students implement a topical algorithm, such as a resolution algorithm), but also short-answer questions (where students provide their answer as a complex Haskell expression representing, for example, a logical formula, an automaton, or a grammar). The approach is similar to Waldmann's AUTOTOOL [11, 13].

We think this approach has the potential to lead to many pedagogical and logistic benefits [2]. We can leverage students' existing programming background to help them learn new mathematical topics, and we can offer students on-demand corrective feedback on their answers. As a digital answer format, Haskell answers are also amenable to automatic and partially-automatic grading workflows.

The approach also has the potential to act as a unified medium for learning and assessment activities across a broad range of topics. Realising that benefit requires carefully designing Haskell exercises to address challenging learning goals such as proof writing. In this paper, we focus on a certain kind of Haskell exercise that partially mimics a traditional proof exercise, contributing evidence for the flexibility of a programming approach.

3 Constructive proofs as programs

In general we have found proof exercises challenging to digitise. However, aspects of certain constructive proofs lend themselves the task. In this section, we lay out the partial analogy we have drawn between Haskell function implementation exercises and certain constructive proofs.

Many important results in our syllabus are of a simple logical form, "for all objects in some class $\mathscr{X}$, there exists an object in some class $\mathscr{Y}$ such that the object has the property P". Examples include proofs of closure properties of formal language classes, or that a certain restricted logical form is universal. Moreover, the most appropriate method of proof is often a direct constructive argument that (1) demonstrates how to build an object in class $\mathscr{Y}$ systematically from the object in class $\mathscr{X}$, and (2) justifies that the constructed object indeed has the property P. Our analogy captures the systematic construction as a Haskell function. Given a data type `X` for objects of class $\mathscr{X}$, and a data type `Y` for objects of class $\mathscr{Y}$, then the systematic construction can be implemented as a Haskell function of type `X -> Y`.

We contend that implementing this function evokes similar skills to writing the constructive argument at the heart of the proof. It requires a working understanding of the 'input' (class $\mathscr{X}$) and 'output' (class $\mathscr{Y}$) objects—enough to manipulate their Haskell representations. It also requires the same rigorous understanding of the details of the appropriate construction algorithm as for a good written description.

We emphasise that the analogy is between the written construction and the Haskell function, not between the entire written proof and the Haskell function. A Haskell exercise does not require the students to justify that the constructed objects will always have the property P. We return to discuss this and other limitations in Appendix A. For now, we merely claim that *some* of what is pedagogically valuable in proof exercises is also present in the Haskell exercises following this analogy. They may be supplemented by other activities addressing any remaining learning goals.

On the other hand, an interactive Haskell environment has countervailing strengths. The Haskell compiler supports and demands syntactic well-formedness. The type system will also detect some conceptual errors (such as confusing sets for their elements). Moreover, Grok affords students this feedback on demand, allowing them to experiment with Haskell and to build confidence in their progress. Moreover, we can configure semantic 'tests', running a sample of inputs through the construction algorithm

```haskell
data Exp
  = VAR  Char
  | NOT  Exp
  | AND  Exp Exp
  | OR   Exp Exp
  | IMPL Exp Exp
  | BIIM Exp Exp
  | XOR  Exp Exp
```

Example: The formula:

$$X \wedge (X \Rightarrow Y)$$

would be expressed in Haskell with the following code:

```haskell
AND (VAR 'X')
    (IMPL (VAR 'X') (VAR 'Y'))
```

Note: This type is suitable for recursive constructions. However for short-answer questions it quickly becomes cumbersome. We provide a string parser utility, so the student could write `parseExp "X & (X => Y)"` here. A type with infix constructors is also possible.

Figure 1: The `Exp` type for propositional logic expressions.

and verify the property P of the output. This falls short of a full verification. However, it can uncover common errors in the informal justification a student has in mind about their function's correctness, prompting them to re-think. Once again, this feedback is available on demand.

Finally, we hypothesise that given our students' backgrounds, a programmatic representation of objects and construction algorithms provides a useful stepping stone to eventually mastering the mathematical formalism (through other activities).

4 Some examples from our course

We explore some example Haskell exercises designed for our course using the analogy to constructive proofs. For each topic, proof exercises comprise a problem description, a data type representing 'input' and 'output' objects, a Haskell program encoding the construction (written by the student), and a suite of tests to verify the construction. We detail these and other notable components for each example.

Our students study propositional logic primarily as a tool for modelling and analysing computational problems. We use a Haskell data type for propositional expressions. The type is outlined in Figure 1.

As part of the study of propositional logic, we explore the expressiveness of connectives. In particular, we explore the diverse *functionally complete* combinations of connectives [10]. To prove a set C of connectives functionally complete, it suffices to show how every formula expressed with a complete set of connectives can be equivalently expressed with C. This construction is readily cast as a Haskell exercise (where the student implements the translation into the restricted form), see Figure 2.

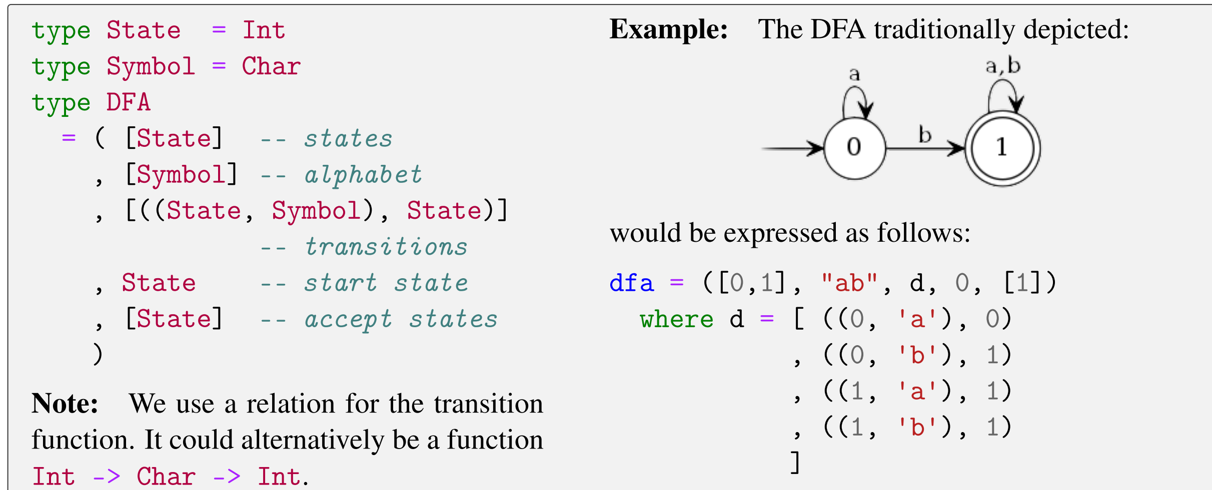
Exercise description: Prove that the set of connectives $\{\Rightarrow, \neg\}$ is functionally complete by writing a function `tr :: Exp -> Exp` that translates arbitrary propositional formulas into equivalent formulas using no other connectives.

Testing: Check that output uses $\Rightarrow$ and $\neg$ only and is equivalent to input.

Haskell answer: A recursive construction.

```haskell
tr :: Exp -> Exp
tr (VAR x)    = VAR x
tr (NOT e)    = NOT (tr e)
tr (IMPL e f) = IMPL (tr e) (tr f)
tr (AND e f)  = NOT (IMPL (tr e) (NOT (tr f)))
tr (OR e f)   = IMPL (NOT (tr e)) (tr f)
tr (BIIM e f) = tr (AND (IMPL e f) (IMPL f e))
tr (XOR e f)  = NOT (tr (BIIM e f))
```

Figure 2: An example propositional logic exercise for a functional completeness construction.



```haskell
type State  = Int
type Symbol = Char
type DFA
  = ( [State]   -- states
    , [Symbol] -- alphabet
    , [((State, Symbol), State)]
               -- transitions
    , State    -- start state
    , [State]  -- accept states
    )
```

Note: We use a relation for the transition function. It could alternatively be a function `Int -> Char -> Int`.

Example: The DFA traditionally depicted:

would be expressed as follows:

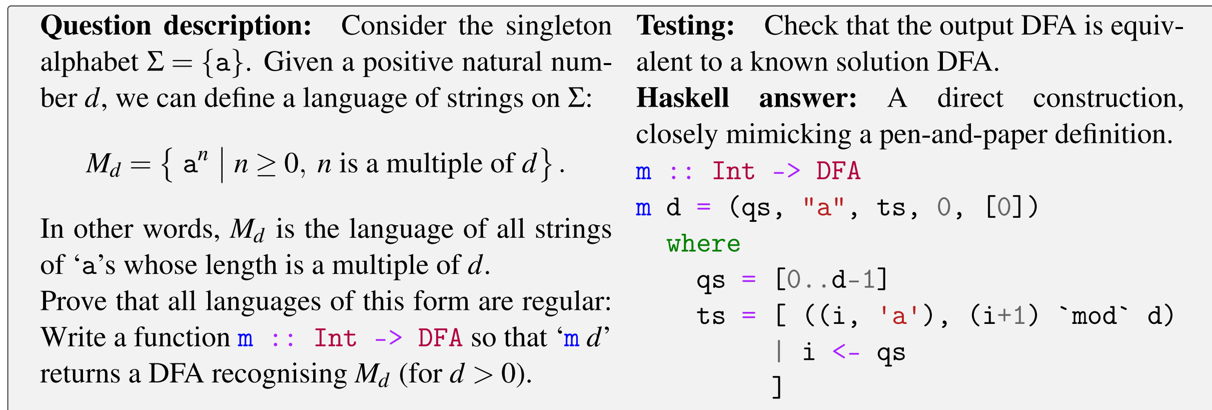
```haskell
dfa = ([0,1], "ab", d, 0, [1])
  where d = [ ((0, 'a'), 0)
            , ((0, 'b'), 1)
            , ((1, 'a'), 1)
            , ((1, 'b'), 1)
            ]
```

Figure 3: The `DFA` type for deterministic finite automata.

Another rich vein of constructive proof exercises come from the theory of regular languages, since they are defined by the existence of some simple automaton or expression. We offer a Haskell data type for regular expressions and for deterministic finite automata (DFAs). The regular expression type is a recursive type similar to the propositional logic type. For brevity we show only the type for DFAs, which directly mirrors the standard mathematical structure. See Figure 3.

A favourite class of exercises gives students a parameterised family of languages and asks them to prove that all languages in the family are regular. It is sufficient to show that for each parameter value, the resulting language has a DFA. This construction is readily cast as a Haskell exercise where the student writes a function to construct the DFA from the parameter. Figure 4 shows an example.

In a full paper, we will provide many further examples in logic and automata theory, along with other examples in formal languages. We will also discuss examples we have conceived but not yet used in class, to illustrate the full range of the analogy.



Question description: Consider the singleton alphabet $\Sigma = \{\mathtt{a}\}$. Given a positive natural number d, we can define a language of strings on Σ:

$$M_d = \left\{ \mathtt{a}^n \,\middle|\, n \geq 0,\ n \text{ is a multiple of } d \right\}.$$

In other words, M_d is the language of all strings of '`a`'s whose length is a multiple of d.
Prove that all languages of this form are regular: Write a function `m :: Int -> DFA` so that '`m` d' returns a DFA recognising M_d (for $d > 0$).

Testing: Check that the output DFA is equivalent to a known solution DFA.

Haskell answer: A direct construction, closely mimicking a pen-and-paper definition.

```haskell
m :: Int -> DFA
m d = (qs, "a", ts, 0, [0])
  where
    qs = [0..d-1]
    ts = [ ((i, 'a'), (i+1) `mod` d)
         | i <- qs
         ]
```

Figure 4: An example DFA construction based on a simple parameterised family of languages.

5 Related work

It has long been realised that a functional programming language can serve as a powerful learning tool for discrete structures and allied topics [4, 12], and we have taken much inspiration from well-known Haskell-based texts on logic [1] and discrete maths [8]. Equally there is a body of work on the teaching of proof techniques firmly grounded in declarative programming (e.g., [5, 9]).

We are not aware of other educators who use a functional programming language, on a single dedicated learning platform, as a *lingua franca* for formative and summative assessment across a wide syllabus, including constructive proof exercises. The closest approach we know of is Waldmann's use of AUTOTOOL [13]. AUTOTOOL offers Haskell exercises spanning an impressive range of topics, and a simple web-based interface. The exercises include implementation tasks and short-answer questions using Haskell as an expression language. Rahn and Waldmann notably demonstrated a Haskell-function based pumping lemma exercise [11], following a similar analogy to our own.

The use of proof assistants (including those based on functional programming, such as Coq and Agda) is orthogonal to the programming-based learning perspective we take. Such tools could potentially fill the gap we leave through our 'construction only' approach to proof. We are not aware of proof assistants used in teaching introductory formal languages. Examples in logic are common (e.g., [5, 7, 9]).

References

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

[2] Matthew Farrugia-Roberts, Bryn Jeffries & Harald Søndergaard (2022): *Programming to Learn: Logic and Computation from a Programming Perspective*. Under review.

[3] *Grok Academy webpage*. `https://grokacademy.org/`. Last accessed Feb 2022.

[4] Peter B. Henderson (2002): *Functional and Declarative Languages for Learning Discrete Mathematics*. In: *Proceedings of the International Workshop on Functional and Declarative Programming in Education*. Available from `https://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/`.

[5] Maxim Hendriks, Cezary Kaliszyk, Femke Van Raamsdonk & Freek Wiedijk (2010): *Teaching Logic Using a State-of-the-Art Proof Assistant*. *Acta Didactica Napocensia* 3(2), pp. 35–48.

[6] *Models of Computation (COMP30026) — The University of Melbourne Handbook (2021)*. `https://handbook.unimelb.edu.au/2021/subjects/comp30026`. Last accessed Feb 2022.

[7] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs*. In: *Verification, Model Checking, and Abstract Interpretation*, *LNCS* 7148, Springer, pp. 24–38.

[8] John O'Donnell, Cordelia Hall & Rex Page (2006): *Discrete Mathematics Using a Computer*. Springer.

[9] Peter-Michael Osera & Steve Zdancewic (2013): *Teaching Induction with Functional Programming and a Proof Assistant*. In: *SPLASH Educators Symposium (SPLASH-E)*.

[10] Emil L. Post (1941): *The Two-Valued Iterative Systems of Mathematical Logic*. Princeton University Press.

[11] Mirko Rahn & Johannes Waldmann (2002): *The Leipzig* `autotool` *System for Grading Student Homework*. In: *Proceedings of the International Workshop on Functional and Declarative Programming in Education*. Available from `https://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/`.

[12] Thomas VanDrunen (2017): *Functional Programming as a Discrete Mathematics Topic*. *ACM Inroads* 8(2), pp. 51–58.

[13] Johannes Waldmann: *Leipzig Autotool webpage*. `https://www.imn.htwk-leipzig.de/~waldmann/autotool/`. Retrieved Jan 2022.

A Limitations of the analogy

In a full paper, we will provide an expanded discussion of the limitations of the analogy we have drawn between constructive proofs and Haskell programs. In particular, the discussion will address the following issues:

- As we have already mentioned, a construction algorithm is only half of a constructive proof—these exercises do not force students to justify that the correctness of their solution.
- To some extent this is mitigated by the provision of automatic test-based feedback, which can help students validate the informal justification they have in mind.
- It may or may not be considered undesirable to encourage students to rely on rapid feedback from a compiler and test cases when writing proofs, compared to rigorous manual verification.
- Our exercises of this form provide an implicit scaffolding to students: By telling them to write a function, we reveal that the proof calls for a construction, rather than having them realise the method of proof for themselves based only on the statement.
- While a Haskell perspective might help students with the challenging task of learning to use a mathematical formalism, Haskell by itself is not easy to learn. As novice Haskell users, our students may face significant barriers to participating in learning and assessment.
- Moreover, the use of Haskell is merely instrumental if our learning goal remains to teach students to use the traditional mathematical formalism. We must ask if the overhead is justified by the actual pedagogical benefits.

We will also address other limitations that are raised through discussions and feedback until the final submission.

B A catalogue of student errors

In a full paper, we plan to include a study of a sample of student responses to an exam question, such as the example demonstrated in Figure 4. We would attempt to categorise the responses.

We expect to find for example that some responses fail to compile due to syntax or type errors; some compile but implement an incorrect construction algorithm in a way that is picked up by a suite of simple test cases; some compile and are incorrect but pass a number of test cases; and some are correct.

During the exam our students are allowed to submit multiple answers. They receive real-time automatic feedback from the compiler but not feedback from testing. Of particular interest would be the trajectory in the correctness of a student's responses over their sequence of submissions. For example, perhaps the student first submits a construction algorithm with a type error indicating a conceptual/semantic error, and then fixes the error and submits again. Perhaps a student first submits a compiling function that would fail on a corner case, and then returns later, realises their mistake, and corrects the error.