Author/s:
Farrugia-Roberts, M;Jeffries, B;Søndergaard, H

Title:
Teaching Simple Constructive Proofs with Haskell Programs

Date:
2022-07-26

Citation:
Farrugia-Roberts, M., Jeffries, B. & Søndergaard, H. (2022). Teaching Simple Constructive Proofs with Haskell Programs. Achten, P (Ed.) Machkasova, E (Ed.) Electronic Proceedings in Theoretical Computer Science Eptcs, 363, (363), pp.54-73. OPEN PUBL ASSOC. https://doi.org/10.4204/EPTCS.363.4.

Persistent Link:
https://hdl.handle.net/11343/322049

Teaching Simple Constructive Proofs with Haskell Programs

Matthew Farrugia-Roberts
The University of Melbourne
Melbourne, Australia
matthew@far.in.net

Bryn Jeffries
Grok Academy
Sydney, Australia
bryn.jeffries@grokacademy.org

Harald Søndergaard
The University of Melbourne
Melbourne, Australia
harald@unimelb.edu.au

In recent years we have explored using Haskell alongside a traditional mathematical formalism in our large-enrolment university course on topics including logic and formal languages, aiming to offer our students a programming perspective on these mathematical topics. We have found it possible to offer almost all formative and summative assessment through an interactive learning platform, using Haskell as a *lingua franca* for digital exercises across our broad syllabus. One of the hardest exercises to convert into this format are traditional written proofs conveying constructive arguments. In this paper we reflect on the digitisation of this kind of exercise. We share many examples of Haskell exercises designed to target similar skills to written proof exercises across topics in propositional logic and formal languages, discussing various aspects of the design of such exercises. We also catalogue a sample of student responses to such exercises. This discussion contributes to our broader exploration of programming problems as a flexible digital medium for learning and assessment.

1 Introduction

We teach *Models of Computation*, a large-enrolment university course on introductory topics in logic, discrete mathematics, formal languages, and computability [14]. Most of our students take the course as part of their degree in computer science or software engineering, wherein they 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 them think and communicate with precision and rigour.

Over several years, we have experimented with a 'programming approach' to learning and assessment activities [8], via the Haskell programming language [10] and the web-based programming education platform Grok Academy [9]. We use Haskell as a stepping stone to traditional mathematical formalism, offering students a programming perspective that leverages their computing background. Haskell also serves as a *lingua franca* for our students to express themselves in exercises spanning our rather broad syllabus in a unified digital learning interface. We have found Haskell problems to be a flexible medium for exercises in many topics beyond programming skill itself.

Traditional exercises of one important class—written proofs—have been difficult to digitise. To this end, we have designed a class of Haskell exercises targeting some of the same skills as certain simple constructive proof exercises. In particular, our students implement the central *construction algorithm* of a constructive proof as a Haskell function. This paper details our approach to partially digitising constructive proof exercises. We contribute: (1) a large catalogue of example *proof-style Haskell exercises* from our course, covering topics in propositional logic and formal languages (Section 6); (2) an analysis of student responses to two questions in a recent exam (Section 7); and (3) a discussion of similarities and differences between written proof exercises and our proof-style Haskell exercises (Section 8).

This work also contributes to our broader initiative of digitising, or 'programmifying', our curriculum with Haskell and Grok Academy, which we have outlined in recent work [8]. The current paper demonstrates in detail how the expressiveness of a programming language affords the design of rich, open-ended digital exercises for non-programming learning goals.

P. Achten & E. Machkasova (Eds.): Trends in
Functional Programming in Education (TFPIE) 2021/22
EPTCS 363, 2022, pp. 54–73, doi:10.4204/EPTCS.363.4

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

Table 1: *COMP30026 Models of Computation*, example semester calendar. Adapted from [8].

2 Course description

We begin by describing our course and our students. 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) [14].

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, with over 500 students in the 2021 offering. Most of our 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 of our students 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.

3 Learning with Haskell and Grok Academy

We have adopted Haskell exercises in learning and assessment tasks. The exercises are delivered through a state-of-the-art web-based programming education platform, Grok Academy (Grok) [9]. In 2021 we retained a small number of 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 give an overview of this approach in recent work [8], and summarise the key elements here.

Since Haskell is not a prerequisite, our semester begins with a self-paced introduction to a minimal subset of features we use. Grok is a suitable platform for this introduction—designed for learning to program, Grok offers a problem-based e-textbook interface. Moreover, Grok is familiar to many of our students who have already used it in their introductory programming courses (Python, C, and/or Java).

However, programming is not one of our direct learning goals. Our students' main use of Grok is for learning *with* Haskell, not learning Haskell. We offer formative and summative assessment through *repurposed* programming problems, specially designed to target our mathematical learning goals.

These exercises typically centre around a *representation*: a Haskell type capturing some mathematical object, such as a propositional logic formula, a relation, or a finite-state automaton. With these representations, we offer students two main kinds of programming problems:

- *Instance problems*, where students answer a traditional short-answer-style exercise by defining an instance of the representation type. For example, we may ask students to design a regular expression for the language of odd-length binary strings, and expect them to respond with a Haskell representation of the expression $(0 \cup 1)((0 \cup 1)(0 \cup 1))^*$.
- *Implementation problems*, where students implement a topical algorithm involving the relevant representations, such as a resolution algorithm for propositional formulas in conjunctive normal form, or an algorithm for determinising a non-deterministic finite automaton.

To configure each problem in Grok, we supply (1) a problem description, (2) skeleton code (including the representation definition, scaffolding, and supporting libraries), and (3) a suite of test cases and associated feedback messages. A student reads the description and writes their answer in Grok's web-based programming interface. They may run the test cases on their answer at any point. Depending on the configuration, the results and/or contents of the test may or may not be revealed to the student. Figure 1 shows the student interface in Grok, with an example *instance problem* using a representation for deterministic finite automata (see Section 6.2).

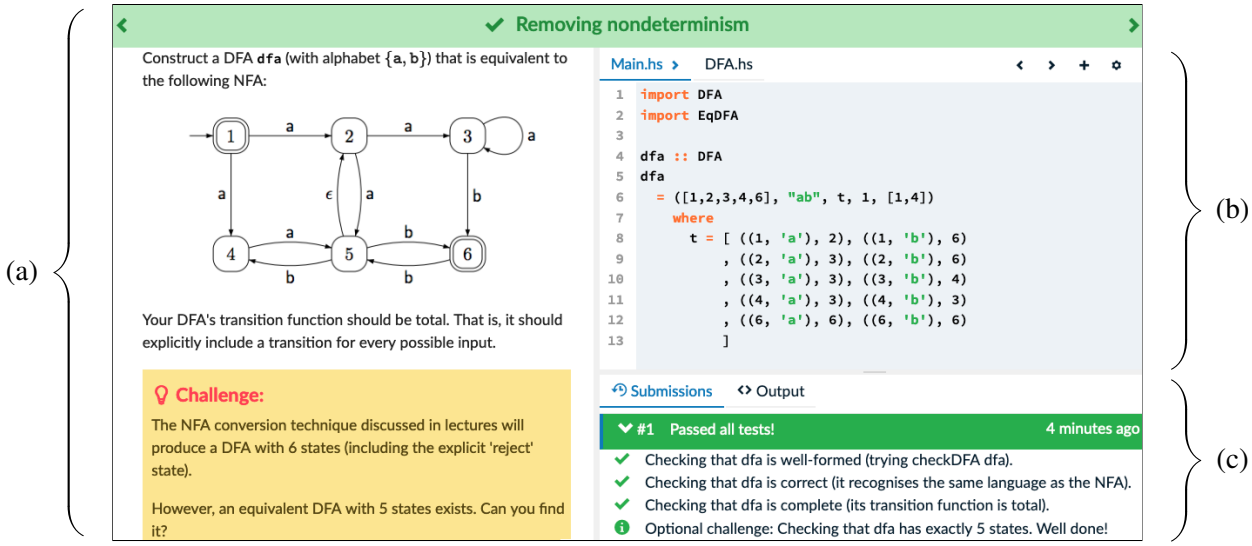


Figure 1: From the student perspective, a programming problem comprises (a) a description (shown: an embedded diagram); (b) a Haskell program editor; (c) feedback messages from a suite of tests. From [8].

The definition of a 'test case' is very flexible, allowing essentially arbitrary Haskell programs to analyse the student's answer. As a result in many cases, especially for instance problems, we can *verify*, rather than merely 'test', a student's response, and sometimes we can also provide helpful contextual feedback. For example, in the automaton exercise in Figure 1, or the regular expression exercise mentioned above, we can run one test to determine if the language of the student's answer precisely matches the desired language. If not, we can provide the student with example strings their solution misclassifies.

We think this approach has the potential to lead to many pedagogical and logistic benefits, provided our students can overcome the concomitant language-related barriers [8]. Realising these benefits requires the careful design of Haskell exercises. In this paper, we expand upon our recent overview [8] with a deep focus on the design of one kind of Haskell exercise. These exercises aim to address the challenging learning goal of *proof-writing* by partially mimicking a traditional proof exercise.

4 Related work

There is a rich tradition for engaging teaching of functional programming, with great ideas for exciting projects and exercises. More recently, the challenges of effective feedback provision for formative assessment, the quest for rapid marking for summative assessment in very large classes, and a worldwide pandemic have spurred developments in reliable auto-marking [20]. This includes the marking of functional programming tasks [1, 13].

It has long been realised that a functional programming language can serve as a powerful learning aid for topics *beyond* programming. Perhaps the most obvious such topic is discrete mathematics. Several authors have pointed to the value that lies, for a student, in the ability to implement and experiment with discrete structures, based on *executable* definitions written in a formalism that is close to familiar mathematical notation [11, 18, 27]. Some authors extend this to topics in logic [7], and there is a body of work on the teaching of proof techniques firmly grounded in declarative programming (e.g., [12, 19]). Moreover, functional programming is sometimes introduced following a "correct-by-construction" philosophy, that is, it is taught hand in hand with program verification [13]. One tool for this is CYP ("check your proof") [4, 5], with support for auto-marking of proof exercises.

Some use functional programming to teach finite-state machines and related topics. For example, Stoughton [25] has used a domain-specific language (DSL) embedded in Standard ML, for the learning of formal language theory. Similarly, FSM [15] offers a DSL embedded in a Lisp variant, for work with state machines. Independently of functional programming, the teaching and learning of automata theory and formal languages have been helped greatly by a range of sophisticated interactive graphical learning tools, of which JFLAP [23, 24] is a prominent example.

The aims of all these tools resemble ours, but we use Haskell *uniformly across our varied syllabus*, on a state-of-the-art interactive learning platform, that (most of) our students already know from their programming education. The use of a functional programming language as a *lingua franca* for formative and summative assessment across such a wide syllabus (including constructive proof exercises) seems rare. The closest approach we know of is Waldmann's use of AUTOTOOL [28]. AUTOTOOL has a simple web-based interface and offers Haskell exercises in non-programming topics such as logic [29], computation [22], and others [29, 30, 31, 32]. Those 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 [22], following a similar analogy to our own.

The use of proof assistants (including those based on functional programming) is orthogonal to our programming-based learning approach. Such tools could potentially fill the gaps we leave through our

'construction only' approach to proof. But we note that the teaching of proof techniques is very different from the application of proof assistants. In our context, where students are trying to learn proof principles, a proof assistant may often *hide* proof details that we would want to expose, such as the details of an inductive step. Examples of proof assistants used in teaching logic are common [12, 17, 19]. We are not aware of examples in introductory formal languages, although there are undoubtedly such cases.

5 Constructive proof exercises and construction problems

In general, we have found proof exercises challenging to digitise as Haskell exercises. However, the algorithmic aspects of certain constructive proofs lend themselves to digitisation. In this section, we lay out the partial analogy we have drawn between written constructive proof exercises and a certain kind of Haskell exercise we call *construction problems*.

Many of the 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 some 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 (1) as a Haskell function. Given a data type `X` representing objects of class $\mathscr{X}$, and a data type `Y` representing objects of class $\mathscr{Y}$, the systematic construction can be implemented as a Haskell function of type `X -> Y`. A *construction problem* is then a Haskell exercise that asks students to implement such a function, framed as a way of proving the original result. Construction problems are thus a kind of *implementation problem* in the terminology of Section 3.

We contend that construction problems evoke similar skills to writing constructive argument at the heart of a pen-and-paper proof, with Haskell serving as an alternative mathematical notation. First, implementing the construction algorithm requires a working understanding of the 'input' (class $\mathscr{X}$) and 'output' (class $\mathscr{Y}$) objects. In particular, it requires enough understanding to manipulate their Haskell representations. Moreover, the implementation requires a rigorous understanding of the details of the construction algorithm. Similar rigour is required as for a precise written description of the algorithm.

We emphasise that the analogy is *partial*. The Haskell function mirrors the written construction part of the written proof, not the entire written proof. In particular, construction problems do not require the students to justify that the constructed objects indeed have the property P.

Nevertheless, this partial analogy suggests that at least *some* of what is pedagogically valuable in written proof exercises may also exist in construction problems. We return to discuss important learning goals missed by construction problems, and other related topics, in Section 8. For now, let us take a tour through our syllabus, looking for constructive proof exercises to convert into construction problems.

6 Example construction problems from our course

Here we focus on the design of construction problems for our course, exemplifying the analogy outlined in Section 5. We list the representations and constructions relevant to propositional logic (Section 6.1), regular languages (Section 6.2), and non-regular languages (Section 6.3), and we showcase several example Haskell exercises in some detail. Throughout, we remark on the design of construction problems based on our experience, and how the exercises fit into our broader Haskell-based approach. Finally, in Section 6.4, we comment on several examples using slight variations of the analogy.

6.1 Examples from propositional logic

Our students study propositional logic primarily as a tool for modelling and analysing computational problems. We use a Haskell data type to represent propositional expressions, as outlined in Figure 2.

```
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:

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

Variants: In some cases, we include `FALSE` and `TRUE` constructors for propositional constants, and/or allow `VAR` labels of types other than `Char`.
It would also be possible to define the type with infix constructors.

Figure 2: The `Exp` type, our representation for propositional logic expressions.

This representation captures the recursive structure of propositional expressions. This structure is convenient when defining recursive constructions. For instance problems, the syntax quickly becomes cumbersome—hence we also provide a string parser utility so that students can write answers such as `parseExp "X & (X => Y)"`.

For the testing/verification of exercises involving propositional expressions, we use Haskell implementations of Wang's algorithm [33] for checking propositional entailment or equivalence, and the Tseitin transform [26] for efficiently encoding expressions in 3-CNF (for processing by a third-party SAT solver). Though testing propositional formulas is NP-complete in general, students typically face small instances. We find that we can comfortably provide automatic feedback within seconds.

A neat feature of our approach is that students get to implement their own version of these tools in early worksheets, giving them at once tools to support and test their own work, and also a deeper understanding of the tools we use in testing.

6.1.1 Functional completeness constructions

As part of the study of propositional logic, we explore the expressiveness of certain logical connectives. In particular, we explore the diverse *functionally complete* combinations of connectives (cf. Post's results [21]). To prove a set of connectives functionally complete, it suffices to show how every formula expressed with a set of connectives known to be complete can be equivalently expressed with the set in question. That constructive argument is readily cast as a construction problem (where the student implements the translation into the restricted form). We outline an example in Figure 3. We revisit this example, denoted FC, in Section 7.

6.1.2 Normal form constructions

A similar class of exercises arises from studying various *normal forms*. Well-known examples include conjunctive normal form, disjunctive normal form, negation normal form, and XOR normal form. We are also free to define our own normal forms (as in FC above). A constructive proof that some set of formulas is a normal form involves translating arbitrary formulas to equivalent formulas in the set. Implementing this translation constitutes a constructive problem.

Exercise description: Prove that the set $\{\Rightarrow, \oplus\}$ of connectives is functionally complete. Do this 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 $\oplus$ only and is equivalent to input.

Haskell answer: A recursive construction.

```
tr :: Exp -> Exp
tr (VAR x)     = VAR x
tr (IMPL e f)  = IMPL (tr e) (tr f)
tr (XOR e f)   = XOR (tr e) (tr f)
tr FALSE       = XOR (VAR 'P') (VAR 'P')
tr TRUE        = IMPL (VAR 'P') (VAR 'P')
tr (NOT e)     = IMPL (tr e) (tr FALSE)
tr (AND e f)   = IMPL (IMPL (tr e) (IMPL (tr f)
                                  (tr FALSE))) (tr FALSE)
tr (OR e f)    = IMPL (IMPL (tr e) (tr FALSE)) (tr f)
tr (BIIM e f)  = IMPL (XOR (tr e) (tr f)) (tr FALSE)
```

Figure 3: An example propositional logic construction problem for a functional completeness result. We return to this example, denoted FC, in Section 7.

6.2 Examples from regular languages

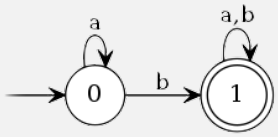
A rich vein of constructive results comes from the theory of regular languages, since they are defined by the existence of some simple automaton or expression, which must usually be constructed.

We offer Haskell data types for deterministic finite automata (DFAs), non-deterministic finite automata (NFAs), and regular expressions. The DFA representation (Figure 4) directly mirrors the standard mathematical '5-tuple' definition. The NFA type is similar, but with multiple start states[1]. Regular expressions use a recursive type (Figure 5). Like for propositional formulas, the type is optimised for expressing recursive algorithms, and we provide a string parser for expressing instances.

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

Variants: We sometimes use other state types. We represent the transition function as an explicit relation. One could use a Haskell function of type `State -> Symbol -> State`.

Example: The DFA traditionally depicted:



is then expressed with the Haskell code:

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

Figure 4: The `DFA` type, our representation for deterministic finite automata.

[1] Our Haskell and mathematical NFAs allow a *set* of start states, as needed in Brzozowski's minimisation method [2, 34].

```haskell
data RegExp
  = Symbol Char
  | EmptyStr
  | EmptySet
  | Concat RegExp RegExp
  | Union  RegExp RegExp
  | Star   RegExp
```

Example: The regular expression

$$0*(1 \cup \varepsilon)$$

could be expressed in Haskell with the following code:

```haskell
Concat (Star (Symbol '0'))
       (Union (Symbol '1') EmptyStr)
```

Figure 5: The `RegExp` type, our representation for regular expressions

When testing exercises involving DFAs, NFAs, and regular expressions, we can provide comprehensive analysis of instances using a suite of Haskell tools for converting NFAs and regular expressions to DFAs (see Section 6.2.3), and then testing equivalence between the resulting DFA and a known solution DFA (by testing if the symmetric difference automaton accepts any strings). This means we can 'test' the correctness of individual DFAs, NFAs, and regular expressions with perfect accuracy, compared to, say, checking a small number of input strings.

Moreover, we can search exhaustively for any strings accepted by the symmetric difference automaton and provide such strings as misclassified examples to the student, as a form of rich, contextual feedback (a similar approach is found in other systems [22, 6]).

The computational complexity of conversion to DFAs and constructing the symmetric difference automaton limit the size of automata and expressions we can analyse (while maintaining responsive automatic feedback). In practice, these limitations do not manifest at the sizes of DFAs and regular expressions we consider but rule out NFAs with more than around 10 states.

6.2.1 Closure property constructions

Regular languages are closed under the regular language operations (union, concatenation, and the Kleene star), as well as the operations of language intersection, complement, reversal, and many others. Proving each such closure property requires an argument that takes automata or expressions characterising arbitrary regular input language(s), and constructs an automaton or expression characterising the output language of the operation. Implementing such constructions makes a suitable Haskell construction problem. Figure 6 shows an example, adapted from an assignment question.

We note that the choice of language representation influences the difficulty of the construction. For example, showing closure under union is trivial for regular expressions and straightforward for NFAs, but requires some care with a DFA representation. Conversely, closure under complement is straightforward in a DFA representation, but for regular expressions or even NFAs, we are unaware of a direct method of construction (not involving, essentially, conversion to a DFA representation).

When it comes to proving these properties for their own sake, one has the choice of representations. In the pedagogical setting, there is value in exploring each method. The choice of representation can be made based on the desired topic or difficulty level. The strategic choice of representations is also a point of interest for our students. While we have not explored this direction in our course, it would possible to allow students to choose a representation as part of their response to a construction problem (by using an appropriate algebraic data type).

Exercise description: Consider the operation *skip* defined by

$$skip(L) = \{xy \mid xzy \in L, x \in \Sigma^*, y \in \Sigma^*, z \in \Sigma\}$$

Informally, the language $skip(L)$ contains all strings obtained from the strings of L by removing any one symbol. For example:

- if $L = \{\varepsilon, \mathtt{ab}\}$, then $skip(L) = \{\mathtt{a}, \mathtt{b}\}$
- if $L = \{\mathtt{a}, \mathtt{b}\}$, then $skip(L) = \{\varepsilon\}$

Show that the class of regular languages is closed under *skip* by writing a Haskell function `skip :: DFA -> NFA` such that

$$\mathscr{L}(\texttt{skip}\ d) = skip(\mathscr{L}(d)).$$

Hint: Build an NFA containing two layers with copies of the initial DFA. Think about how to connect the layers and how to define the start and accept states.

Testing: For several inputs, check that output NFA is equivalent to known correct DFA.

Haskell answer: Following the hint:

```
skip :: DFA -> NFA
skip d@(qs, xs, ts, q0, as)
  = ( qs++qs'
    , xs
    , ts++ts'++ts''
    , [q0]
    , as'
    )
  where
    -- With helper fn. 'renameDFA',
    -- make 2nd layer with distinct
    -- states
    rename
      = (+(1+(maximum (map abs qs))))
    (qs', _, ts', _, as')
      = renameDFA rename d
    -- Epsilon transitions between
    -- layers implement the skip:
    ts''
      = [ ((q, epsilon), rename r)
        | ((q, x), r) <- ts
        ]
```

Figure 6: An example construction problem based on a closure property of regular languages.

6.2.2 Language family constructions

A favourite class of constructions is to give students a parameterised family of languages and ask them to prove that all languages in the family are regular. It is sufficient to construct a DFA (for example) for each parameter, generalising the traditional task of designing a DFA for a particular language. This exercise is readily cast as a construction problem, where the student writes a function to build the DFA based on the parameter. Figure 7 shows an example (LA) adapted from a final exam.

Many variations of this example are possible. A challenging construction exercise we used in a previous assignment highlights one of the advantages of using a single medium for exercises across our broad syllabus. We designed a family of languages parameterised by propositional logic formulas. Given a formula, the corresponding language consisted of strings encoding satisfying truth assignments for the formula. Through a recursive DFA construction algorithm, students were able to prove that this family of languages is regular. The recursive construction was also an opportunity for students to use, as helper functions, constructive algorithms for product and complement DFAs, built in a previous worksheet.

6.2.3 Representation conversion constructions

A staple of an introductory course on formal languages are the constructive proofs showing that DFAs, NFAs, and regular expressions all capture the same class of languages, since for every regular expression

Exercise 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.

```
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 7: An example DFA construction based on a simple parameterised family of languages. We return to this example, denoted LA, in Section 7.

there is a corresponding DFA, and so on. Some of these construction algorithms are well-suited as complex implementation exercises. In particular, we typically include an exercise to implement an NFA determinisation algorithm in a worksheet, and we have in the past included, in an assignment, a guided implementation of Brzozowski's algorithm for converting regular expressions directly to DFAs [3].

These same tools form part of our testing apparatus for regular language exercises, which first converts a student's NFAs or regular expressions to DFAs, as discussed above. Once again, students can take part in the building of tools that we (and they) use to analyse their work.

These conversions are also useful as 'helper functions' in other exercises. For example, after implementing the NFA determinisation function, our students are a short step away from implementing Brzozowski's double-reversal method for DFA minimisation [2, 34].

Another construction algorithm of interest is the 'union-free decomposition' of regular expressions. For any regular language, there is always a representation of the language as a union of a finite number of regular expressions which, themselves, do not contain the union operation [16]. In a challenging construction problem, which we have used in a final exam, we explain this representation to students, and then ask them to write the conversion algorithm. We detail this example in Figure 8.

6.3 Examples from non-regular languages

In the study of more complex languages, our students complete exercises involving Haskell representations of context-free grammars (CFGs), (deterministic) pushdown automata ((D)PDAs), Turing machines, and other models of computation. For brevity, we omit a detailed description of the representations here—suffice it to say that they are similar to the DFA representation shown in Figure 4, again mirroring the standard tuple-based mathematical representation of these objects.

With instances of these models, we quickly move into the realm of the undecidable. It is not possible in general to decide if a student's model recognises a particular language. We have a suite of emulation tools, and can fall back on string-based testing and, if necessary, manual inspection, which is usually sufficient (students do not often submit pathological or even large instances).

Most of our non-regular language exercises involve the creation or analysis of specific instances (in the case of analysis, we can often design the exercise so that correctness is decidable). However, we have designed a small number of construction problems, as we outline below.

Exercise description: A regular expression is union-free if it does not use the union operator. A union-free decomposition of a regular language R is a finite list of union-free regular expressions $r_1, \ldots, r_k$ such that $R = \mathscr{L}(r_1) \cup \cdots \cup \mathscr{L}(r_k)$. Prove that every regular language has a union-free decomposition. Do this by writing a Haskell function `uf :: RegExp -> [RegExp]` that takes a regular expression `r` and produces a union-free decomposition of $\mathscr{L}(\texttt{r})$.
Hint: The following rules for regular expressions may be useful.

$$\mathscr{L}\big(r(r_1 \cup \cdots \cup r_n)\big) = \mathscr{L}\big(r\,r_1 \cup \cdots \cup r\,r_n\big)$$
$$\mathscr{L}\big((r_1 \cup \cdots \cup r_n)r\big) = \mathscr{L}\big(r_1\,r \cup \cdots \cup r_n\,r\big)$$
$$\mathscr{L}\big((r_1 \cup \cdots \cup r_n)*\big) = \mathscr{L}\big((r_1 * \cdots r_n*)*\big)$$

Haskell answer: Following the hint:

```
uf :: RegExp -> [RegExp]
uf (EmptyStr)   = [EmptyStr]
uf (EmptySet)   = [EmptySet]
uf (Symbol x)   = [Symbol x]
uf (Union r s)  = uf r ++ uf s
uf (Concat r s)
  = [Concat a b | a <- uf r, b <- uf s]
uf (Star r)
  = [Star (cat [Star s | s <- uf r])]
```

Note: A helper function, equivalent to `cat = foldr1 Concat` was provided to students with documentation.

Testing: For various inputs, check that the output decomposition is equivalent to the input regular expression.

Figure 8: An example construction problem based on a representation of regular languages.

6.3.1 Closure property constructions

As with regular languages, one class of construction exercises comes from closure properties of higher language classes. For example, students can write the construction by which one can see that the classes of context-free and decidable languages are closed under the regular operations, or other operations such as reversal, and intersection with regular languages.

6.3.2 Representation conversion constructions

Though we are yet to deploy examples in this vein, there are also several representation conversion results for higher language classes that may be suitable for construction problems. For example, one could consider conversions between CFGs and PDAs, or various Turing-equivalent models, and converting CFGs to Chomsky normal form. These detailed constructions could be worth students' implementing as an assignment challenge or in a guided worksheet.

One class of conversions that we have considered is to convert regular models into CFGs, PDAs, and TMs. A student can prove that the language classes associated with these models contain the regular languages through such constructions. Most of these constructions are quite straightforward (e.g., making no use of the PDA stack or TM tape), but may still be worthwhile for students.

A more challenging variation is to apply a restriction on the construction. For example, it is possible to construct a deterministic PDA for any regular language (say, represented as a DFA) using only three PDA states (regardless of the DFA size) by utilising the stack. We detail this example in Figure 9. Similar constrained constructions are available for TMs and may be possible for other models. To prove these results, students can implement the constrained construction algorithms in Haskell.

As with many construction problems, but particularly in this case, we must take care to avoid an overload of low-level Haskell details. It is important to design exercises with clean implementations.

Exercise description: Prove that any regular language can be recognised by a deterministic PDA (DPDA) with just three states. To do this, write a Haskell function, `dfa2pda :: DFA -> DPDA` that converts an arbitrary DFA to an equivalent DPDA with exactly three states.
Hint: While there is a limit on the size of the PDA's state machine, there is no limit on the size of the PDA's stack alphabet.

Haskell answer: From the start state, load the second DFA state onto the stack while consuming the first symbol. Transition to an accept or non-accept state in the DPDA depending on where you would be in the DFA. Carry out the remaining transitions using the symbol on the stack to represent the state in the DFA, and the DPDA state to represent whether the DFA is ready to accept or reject.

```
dfa2pda :: DFA -> DPDA
dfa2pda (qs, as, ts, q0, fs)
  = ([0, 1, 2], as, qs, ts', 0, fs')
where
  accepts q
    = if (elem q fs)  then 2 else 1
  fs'
    = if (elem q0 fs) then [0,2] else [2]
  ts'
    =  [ ( (accepts b, a, b)
         , (accepts b', b')
         )
       | ((b, a), b') <- ts
       ]
    ++ [ ( (0, a, eps)
         , (accepts b', b')
         )
       | ((b, a), b') <- ts
       , b == q0
       ]
```

Figure 9: An example construction problem based on a non-regular model of computation.

6.4 Variations of construction problems

So far we have discussed construction problems based on constructive algorithms of the form `X -> Y` for some representation types `X` and `Y`. In this section we collect notes on several variations on this theme.

6.4.1 Instantiation as scaffolding

To each construction problem correspond many simpler problems of the form 'for this *given* object of class $\mathscr{X}$, construct the corresponding object of class $\mathscr{Y}$'. While the general construction problem is a kind of implementation problem, that is, a Haskell exercise with answer type `X -> Y`, these simpler problems correspond to instance problems—Haskell exercises with answer type `Y`.

When designing an assignment or exam, we can often calibrate the level of difficulty, as well as target a different set of learning outcomes, by changing between instance problems and implementation problems. Moreover, a useful design pattern is to run instance problems and implementation problems together. For example, in one multi-part exam question, students might be presented with one or two examples of a construction in the form of instance problems and then be asked in a final part to implement the general construction. The instances serve as a 'warm-up' for the general algorithm, and also give students credit for time spent considering examples as they prepare to provide a general answer in the final part.

6.4.2 Proof-based framing of instance problems

There are many places in our course where we might use the term 'constructive proof' in a slightly different sense than above, not referring to a particular general construction algorithm, but to the construction of a witness to an existential claim (or a counterexample to a universal one).

By a similar analogy to that underlying construction problems (of the implementation type), we can draw an analogy between these instance constructions and instance problems. The distinction is that between a Haskell function with parameters, and a constant Haskell function. We often frame an instance problem this way, as a kind of 'proof', to more closely relate our Haskell instance exercises to the mathematical topics of our syllabus. For example, rather than describing an exercise involving CFG ambiguity as "give an example string for which this CFG permits at least two distinct parse trees", we might say "prove that this CFG is ambiguous by giving a string for which ..." Many other examples of this form exist, across propositional logic, regular languages, non-regular languages, and also predicate logic and discrete mathematics.

Once again, this analogy is *partial*, since students usually are not required to carefully justify the correctness of their examples. However, we can often verify correctness with appropriate tests if the problems are carefully designed so that correctness remains decidable.

6.4.3 Proof-based framing of algorithm implementation problems

In a sense that is relevant to the topics of our course, all of the implementation problems we study, whether framed as constructive proofs (such as the examples earlier in this section) or as algorithms (such as a DFA minimisation algorithm, Wang's algorithm, the Tseitin transform, or a propositional resolution algorithm), are a kind of existence proof that certain problems are decidable (and inhabit a certain complexity class).

For example, while we usually frame the students' implementation of the Tseitin transform in an early worksheet as the completion of a useful tool for computing with propositional logic representations, it is equally an existence proof that a (polynomial-time) algorithm exists to convert an arbitrary propositional formula to 3-CNF. This particular algorithm also serves as a (polynomial-time) reduction between the Boolean satisfiability problem for arbitrary formulas, and the 3-SAT problem. Thus it provides a nice opportunity to link several topics within our broad course.

Again we note that the analogy is *partial*, as students are not normally asked to carefully justify their implementation's correctness (or complexity).

6.4.4 Higher-order constructions

In logical terms, the construction problems we have examined in detail above correspond to a constructive proof of a universally quantified existence result. Similarly, appropriately-framed instance problems correspond to a constructive proof of a simple existence result.

The possibility of extending this analogy is suggested: Could we make a Haskell exercise out of the proof of a higher-order result? For example, a result of the form "for all $x \in \mathscr{X}$, there exists $y \in \mathscr{Y}$, such that for all $z \in \mathscr{Z}$, there exists $u \in \mathscr{U}$, such that the property $P(x,y,z,u)$ holds."

For example, a proof that a language does not have the pumping property for regular languages has exactly the above form. Such a proof, together with the well-known pumping lemma for regular languages, is often sufficient to show that a language is non-regular. The same holds for a proof that a language does not have the pumping property for context-free languages. Pumping lemma exercises are regarded by many of our students as a very challenging part of our course, perhaps partly because of

their mathematical complexity. If some Haskell-based exercise could assist the students in constructing a pumping lemma proof, this could be pedagogically valuable.

It would be possible to ask students to implement functions that play the role of the interacting agents in a query-based proof of the pumping property for a language, for example. This approach to the pumping property would be similar to the 'pumping lemma game' available in JFLAP [23]. A similar, Haskell-based approach has already been developed for AUTOTOOL [22].

One concern is that the specification of the Haskell-based exercise could retain the complexity of the mathematical approach, due to the necessary use of higher-order functions and other challenging language features. If this analogy is to be extended, it will not remove the inherent logical complexity of the results to be studied.

7 Student performance in high-stakes assessment

As we have gradually moved to our new assessment regime, eventually delivering *all* assessment through Grok, we have naturally wondered what consequences (predicted or unforeseen) there might be for students. The concern was never that students might be uncomfortable with online assignment submission and online exams—if anything, there seems to be a growing percentage of students who feel uncomfortable with pen-and-paper assessment and instead value the better writing and editing features offered by online tools. Instead our concern has been that we are forcing students to communicate their knowledge in a new and unforgiving notation, namely Haskell. There are arguable advantages and disadvantages involved, for learning, feedback provision, learning support, student engagement, marking, and course management generally [8]. So how do students respond to the change?

We do not have data from surveys asking students directly (apart from standard student experience questionnaires, to which students continue to respond positively about the course generally). Grok, however, provides us with logs that allow us to explore students' activity, for example during exams. This gives us some insight into whether Haskell ends up being a significant barrier to students expressing themselves, in particular in their answers to more difficult problems, such as proof constructions.

In this section, we sample student responses to two exam questions. These are the questions we labelled FC and LA in Section 6. High-stakes assessment is arguably the most suitable context for this analysis. During an exam, students can submit answers to a problem repeatedly until they are happy with their answer—only the latest submission counts. In the exam setting they receive real-time feedback from the compiler, and, where relevant, additional automated feedback about the well-formedness of their answer, beyond syntax. They do not see the results from any of our correctness tests.

7.1 The FC example from Section 6.1

In the exam, 517 students submitted answers to this question, while 38 did not submit. Students who did submit made 2.57 submissions, on average. In the following, we analyse each student's *final* answer only, in case they made multiple submissions. Of the 517 answers, 483 passed the well-formedness tests and were then marked for (partial) credit according to their degree of correctness. The remaining 34 failed to compile. According to GHCI's feedback to the 34 students, the errors were distributed as shown in Table 2, second column.

More careful scrutiny reveals that almost all (12 of 13) type errors were the result of incorrect use of parentheses in function applications. In essence, 12 students wrote something of form `a b c` when they meant `a (b c)`. Many scope errors came from code such as

Issue	FC (Fig. 3)	LA (Fig. 7)
Indentation problem	3	10
Syntax error	2	16
Variable scope error	14	7
Hole ('_') used as expression	1	0
Nonlinear pattern	1	0
Pattern overlap	0	1
Bad enumeration	0	2
Type error	13	6
Generation of ill-formed DFA	—	22
Failure to terminate	0	2

Table 2: Number and kind of responses received by students regarding ill-formed submissions.

```
tr FALSE = tr (AND (VAR x) (NOT (VAR x)))
```

which shows a fair understanding of the propositional logic problem. But the variable scope error prevents such understanding from being recognised, even by sophisticated auto-marking. For this reason, we mark all non-compiling submissions manually. That way (possibly partial) marks can still be awarded for plausible code. There were three cases of GHCI flagging an indentation error; in all cases, these were caused by students using a layout similar to that in Figure 3, but leaving one or more cases unfinished, with no text after a defining '='.

To summarise, student performance on this question does not suggest that Haskell has "gotten in the way", except for a small number of students who struggle with the role of parentheses in a language where simple juxtaposition is used for function application.

7.2 The LA example from Section 6.2

In the exam, 499 students submitted answers to this question, while 56 did not submit. Students who did submit made 2.39 submissions, on average. Again the analysis we perform of the submitted answers involves only each student's final answer. Of the 499 answers, 42 failed to compile. GHCI feedback suggests the errors were distributed as listed in the third column of Table 2.

Altogether 66 students would have received immediate feedback to warn them that something about their answer was wrong. Namely, in addition to the 42 with compiler errors, two were warned about termination issues, and another 22 failed to generate well-formed DFAs. In the last case, a message, auto-generated by us, explained in what way the DFA was ill-formed (perhaps it was not deterministic, or it used letters outside its alphabet, or states outside its state set). For 22 students, it appears this was not sufficient help, or alternatively, some students had insufficient time left to fix the issue.

As the feedback generated at submission time was based on a single test case ($d = 5$), there may be cases of ill-formedness (and non-termination) that go undetected. Indeed, the full set of test cases identified another 11 cases of ill-formed DFAs being generated—almost always for the corner case $d = 1$.

The 10 indentation errors are mostly because of incorrectly placed `where` clauses. Of the 16 syntax errors, 10 show difficulty with list comprehension syntax, like pointing arrows in generators the wrong way. The scope errors are all from expressions involving list comprehension. This kind of error, where a student uses a variable outside its scope, should not necessarily be taken as evidence that Haskell has hindered expression; rather these cases mirror the identical mistake we see when students use mathematical notation, such as set comprehension, leaving variables free.

We mentioned that 56 students did not submit an answer. In some cases, the distinction between "no attempt" and "syntax error" is not clear. For example, when a student submits

```
makeDFA d = (
```

we classify that as a syntax error, but in all likelihood, it is the result of a student deciding not to attempt the question (and submitting the snippet nevertheless). Of the 16 syntax error cases mentioned in Table 2, third column, five are arguably in this category, that is, they do not really suggest a lack of knowledge of Haskell syntax. Likewise, three of the 22 ill-formed DFAs come about because a student simply submits the bare code that was provided as scaffolding for the question—evidence that the student simply decided to give up this question.

Hence the number of student submissions (in the case of this particular question) that point to a struggle with Haskell notation is perhaps more accurately given as 34, but in any case, it is in the vicinity of 7% of submissions. The number of students who appear to give up the question (whether for lack of time or ability) is 64, that is, around 12%—not unlike what we would previously see in a typical pen-and-paper question of this kind.

To summarise, submitted answers for this question suggest that 5–10% of our students begin to struggle with Haskell once a question calls for a combination of language features including recursive definition, `where` clauses and list comprehension. To preserve fairness, manual marking seems unavoidable for answers that get rejected by GHCI.

8 Discussion

We turn to discuss the various pedagogical and logistic dimensions of the choice between written proof exercises and our proof-inspired Haskell implementation exercises.

8.1 Pedagogical evaluation

In terms of the pedagogical value of these two learning activities, a comparison must account for the skills which each task will exercise for students, and the alignment of these skills with the intended learning outcomes. Section 8.1.1 discusses aligned skills neglected by our Haskell exercises, and Section 8.1.2 discusses additional skill requirements introduced by the change in exercise format.

8.1.1 Additional skills exercised by written proofs

As we have outlined above, we believe that well-designed Haskell implementation exercises can bring out many of the same kinds of skills as the writing of a detailed written description of a construction algorithm. These include the ability to specify the construction itself with a high level of precision, drawing on a detailed understanding of the formal objects involved, and of the analysis and creation techniques embodied by the construction. These all may be considered intended learning outcomes in many maths-based classes such as our own.

As we produce a partial analogy, it is clear that the skills involved in analysing and justifying one's construction are not exercised directly in the same manner as in a written proof. If this is considered an important learning outcome, it may be necessary to augment Haskell-based activities with opportunities for students to practice and receive feedback on written justifications, as we do in our course.

It is worth considering the role that rapid corrective feedback from automated tests could play as a kind of substitute for explicit justification tasks. We suppose a student has an *implicit* justification in mind

guiding the design of their constructive algorithm. When they receive feedback from tests, for example that their construction fails in some case, they may adapt their implicit justification to resolve an error in their approach. The student is exercising their justification skills in a minor way, though not the skill of formalising and explicating their justification. On the other hand, unlimited rapid feedback may lead students to somewhat mindlessly tweak their implementation, using a kind of 'trial-and-error' approach, without maintaining an internal justification. Future work could investigate how students respond to feedback, for example with a think-aloud study.

Another class of skills that may be neglected in Haskell-based exercises arise due to the structure imposed by our interface. In designing a test suite we usually make assumptions about the Haskell type of the student's answer. We must then explain to the student which type we expect as part of the exercise description. Thus the student does not get to exercise the skill of taking a proposition and thinking for themselves about what sort of constructive proof would be appropriate to establish this proposition. This gap could potentially be addressed through careful Haskell exercise design, such as by offering students a choice of types with which to respond using a suitable algebraic data type. However, the most direct response may be to offer students an opportunity to practice open-ended written proofs.

The skill of writing fluent and clear mathematical arguments in mathematical notation is clearly not directly accessible through Haskell exercises. If this skill is an intended learning outcome, then perhaps written exercises should be emphasised. It is possible to practice communication skills by writing clear Haskell programs as answers to Haskell exercises, but assessing these dimensions of Haskell programs falls beyond the remit of the compiler and simple automatic tests. Anyway, the premise of using Haskell as a bridge to a mathematical formalism and notation suggests that we should eventually help our students *cross* the bridge into the more conventional language of mathematics.

8.1.2 Additional skills demanded by Haskell exercises

The main skill demanded by Haskell exercises in addition to those required to complete a written proof is, of course, the ability to express one's ideas in a functional programming language. Completing Haskell-based exercises requires students to become reasonably proficient in Haskell. At times, students are also required to provide some language-level details in the specification of the construction which would be below the level of abstraction appropriate for a clear written proof. Furthermore, taking advantage of the feedback provided by the compiler also requires the skill of interpreting Haskell error messages.

To minimise these barriers, we take care to design and select Haskell exercises that use a minimal subset of language features, and have answers that do not require much low-level detail. This sometimes involves carefully scoping problems for students to implement a small part of a larger program, with the aid of helper functions that present a clear abstraction hiding complicated parts of the construction (noting that adding novel helper functions to an exercise itself adds cognitive load for students). The overall impact of these barriers on student outcomes in our course requires further evaluation.

8.2 Digital formative and summative assessment

Rapid compiler-based and test-based feedback have the potential to enhance student learning of the exercised skills, compared to written proof exercises. With traditional proof exercises, providing rapid feedback to students at scale and on-demand is infeasible. In our case, this kind of environment can only be provided to students during a fraction of time spent in tutor-lead weekly tutorials (that fraction when the tutor is available to interact directly with a small group of students, and the syllabus allows this time to be spent on proof-writing exercises, as opposed to some other intended learning outcome).

In comparison, Grok affords students flexible, on-demand, instant corrective feedback from our pre-designed test suites. No doubt this feedback helps students notice gaps or errors in their specifications of constructions, and train the related skill.

Aside from providing rapid feedback in formative assessment, we also provide compiler-based and limited test-based feedback to students during high-stakes tests such as final exams, and we allows students to fix such errors and resubmit their answers an unlimited number of times. We believe that this policy can increase the validity of assessment by helping students filter out small errors such as typos, syntax errors, and well-formedness errors from their answers before they are evaluated.

When it comes to marking student responses in a digital format compared to marking written proofs, we find that Haskell exercises allow for more flexible and often more efficient marking workflows compared to written proofs. Note that as a baseline, it is always possible to print and manually mark student programs, as if they had been written on paper in the traditional manner. At the other extreme, high-quality, fully automated marking of the kinds of complex Haskell programs we are discussing in this paper would require more sophisticated analysis programs than we typically have the resources to commission for our course. The key point is that, with a student's digital response, we are free to blend manual marking work and automation in a flexible way, and even to change this mix during marking. We can automate tasks that are well-suited to automation, such as compilation and testing of student programs (cf. a tutor poring over a student's handwritten construction, attempting to run it through her own mental logical compiler and correctness tests). And we can manually handle the remainder: the parts that may call for human judgement, such as evaluating non-compiling solutions for partial credit, and judging student programs against a marking scheme that takes more than test correctness into account. Importantly, for most instance problems we can also simplify the manual marking residue by *automatically* clustering incorrect responses. For example, submitted formulas or regular expressions can be grouped by size and/or semantic equivalence, requiring a human marker to determine the merits only of *representative* solutions—something of considerable value when enrolment numbers are large.

9 Conclusion

The objective behind our use of Haskell as an assessment medium in a non-programming course has been to give computing students a bridge to a mathematically demanding syllabus. We saw potential benefits, both logistic and pedagogical, of a digitisation of elements of the course [8]. Auto-marking especially is attractive, though it is far from clear that depth, quality and reliability of assessment can be maintained in the transition. We now conduct practically all assessment digitally, through Haskell.

Harnessing Haskell to do the job of traditional online question types, such as multiple-choice, is straightforward, once students have completed a short intensive introduction to the language. But we would like to also use Haskell for various open-ended (or "constructed response") questions. In this paper we have shown how we administer certain proof-style problems. Our approach integrates seamlessly with the other question types we use—all are delivered on one interactive learning platform using Haskell as the assessment medium. We have provided many examples of constructive-proof problems that we have used in assessment, including exams.

These more sophisticated problem types are well-suited for testing higher-order cognitive skills. But they also tend to assume a greater mastery of Haskell, something that is not actually expressed as an intended learning outcome in our course. This raises the question of whether Haskell might present a barrier for some students who would otherwise have no difficulty expressing themselves in traditional mathematical language. We have very limited data on which to base an answer, and more research is

needed. However, the analysis presented in Section 7 suggests that 5–10% of our students do find it hard to express themselves well in Haskell when an answer calls for a combination of Haskell features. Given the logistic benefits of the programming approach, we will continue to calibrate assessment tasks to find the right balance between medium and expressiveness.

Acknowledgements

We thank our teaching colleagues Anna Kalenkova, Bach Le, Billy Price, Rohan Hitchcock, and the rest of the COMP30026 teaching team. We thank Kevin Kappelmann, TFPIE2022 conference attendees and our anonymous reviewers for helpful discussions and suggestions.

References

[1] Joachim Breitner, Martin Hecker & Gregor Snelting (2017): *Der Grader Praktomat*. In O. J. Bott et al., editors: *Automatisierte Bewertung in der Programmierausbildung*, *Digitale Medien in der Hochschullehre* 6, Waxmann Verlag GmbH, Münster, Germany, pp. 159–172. Available at `https://www.waxmann.com/automatisiertebewertung/`.

[2] Janusz A. Brzozowski (1962): *Canonical Regular Expressions and Minimal State Graphs for Definite Events*. In: *Proceedings of the Symposium on Mathematical Theory of Automata*, *MRI Symposia* 12, Polytechnic Institute of Brooklyn, NY, pp. 529–561.

[3] Janusz A. Brzozowski (1964): *Derivatives of Regular Expressions*. *Journal of the ACM* 11(4), pp. 481–494, doi:10.1145/321239.321249.

[4] *Cyp—Checker for "Morally Correct" Induction Proofs about Haskell Programs*. Available at `https://github.com/noschinl/cyp`. Retrieved Apr 2022.

[5] *Cyp—for Use with Leipzig Autotool*. Available at `https://gitlab.imn.htwk-leipzig.de/waldmann/cyp`. Retrieved Apr 2022.

[6] Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan & Björn Hartmann (2015): *How Can Automatic Feedback Help Students Construct Automata? ACM Transactions on Computer-Human Interaction* 22(2), doi:10.1145/2723163.

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

[8] Matthew Farrugia-Roberts, Bryn Jeffries & Harald Søndergaard (2022): *Programming to Learn: Logic and Computation from a Programming Perspective*, doi:10.1145/3502718.3524814. To appear in *Proceedings of the 27th ACM Conference on Information Technology in Computer Science Education*, Dublin, July.

[9] *Grok Academy webpage*. `https://grokacademy.org/`. Retrieved Feb 2022.

[10] *Haskell webpage*. `https://www.haskell.org/`. Retrieved Apr 2022.

[11] 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/`.

[12] 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.

[13] Kevin Kappelmann, Jonas Rädle & Lukas Stevens (2022): *Engaging, Large-Scale Functional Programming Education in Physical and Virtual Space*. *Electronic Proceedings in Theoretical Computer Science* 363, Open Publishing Association, pp. 93–113, doi:10.4204/EPTCS.363.6.

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

[15] Marco T. Morazán & Rosario Antunez (2014): *Functional Automata: Formal Languages for Computer Science Students*. In J. Caldwell, P. Hölzenspies & P. Achten, editors: *Proceedings of the Third International Workshop on Trends in Functional Programming in Education (TFPIE 2014)*, *EPTCS* 170, pp. 19–32, doi:10.4204/EPTCS.170.2.

[16] Benedek Nagy (2004): *A Normal Form for Regular Expressions*. In: *Supplemental Papers for the 8th International Conference on Developments in Language Technology*, CDMTCS Research Report, pp. 53–62.

[17] 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, doi:10.1007/978-3-642-27940-9_3.

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

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

[20] Joseé Carlos Paiva, Joseé Paulo Leal & Álvaro Figueira (2022): *Automated Assessment in Computer Science Education: A State-of-the-Art Review*. *ACM Transactions on Computing Education* 22(3), pp. 34:1–34:40, doi:10.1145/3513140.

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

[22] 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/`.

[23] Susan H. Rodger: *JFLAP webpage*. `https://www.jflap.org/`. Retrieved Jan 2022.

[24] Susan H. Rodger, Bart Bressler, Thomas Finley & Stephen Reading (2006): *Turning Automata Theory into a Hands-On Course*. In: *Proceedings of 37th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, pp. 379–383, doi:10.1145/1121341.1121459.

[25] Alley Stoughton (2008): *Experimenting with Formal Languages Using Forlan*. In: *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*, pp. 41–50, doi:10.1145/1411260.1411267.

[26] G. S. Tseitin (1968): *On the Complexity of Derivation in the Propositional Calculus*. In A. O. Slisenko, editor: *Studies in Constructive Mathematics and Mathematical Logic*, Part II, pp. 115–125, doi:10.1007/978-3-642-81955-1_28.

[27] Thomas VanDrunen (2017): *Functional Programming as a Discrete Mathematics Topic*. *ACM Inroads* 8(2), pp. 51–58, doi:10.1145/3078325.

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

[29] Johannes Waldmann (2014): *Automated Exercises for Constraint Programming*. In: *28th Workshop on (Constraint) Logic Programming (WLP 2014)*, pp. 66–80.

[30] Johannes Waldmann (2015): *Automatisierte Bewertung und Erzeugung von Übungsaufgaben zu Prinzipien von Programmiersprachen*. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*.

[31] Johannes Waldmann (2017): *Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen*. In: *Proceedings of the Third Workshop "Automatische Bewertung von Programmieraufgaben" (ABP2017)*, 2015.

[32] Johannes Waldmann (2017): *How I Teach Functional Programming*. In: *Workshop on Functional Logic Programming (WFLP 2017)*.

[33] Hao Wang (1960): *Toward Mechanical Mathematics*. *IBM Journal of Research and Development* 4(1), pp. 2–22, doi:10.1147/rd.41.0002.

[34] Bruce W. Watson (1995): *Taxonomies and Toolkits of Regular Language Algorithms*. Ph.D. thesis, Eindhoven University of Technology.