Teaching simple constructive proofs with Haskell programs

presentation by Matthew Farrugia-Roberts
joint work with Harald Søndergaard

The University of Melbourne
Melbourne, Australia

TFPIE 2022

Our course: Models of Computation

Introduction to logic, discrete math, formal languages, and computability.

Students:

- Over 500 students (2021 semester)
- Mostly computer science and software engineering students
- Programming background (Python, C, Java, some students Haskell)
- Diverse mathematics backgrounds

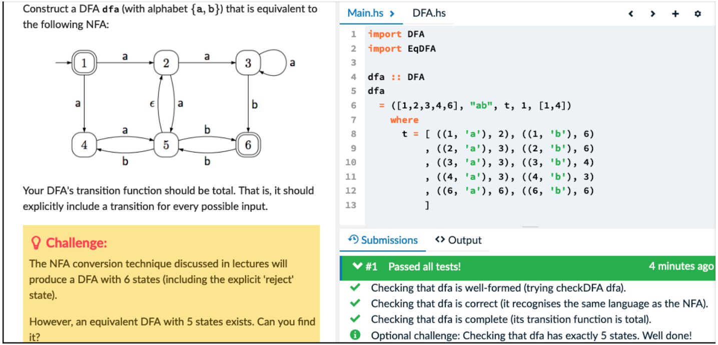
Idea: Leverage programming background as a bridge to the maths concepts...

Week	Topic	Haskell exercises for learning and assessment	
1	Introduction	Introduction to Haskell basics (functions, recursion, lists, algebraic data types)	
2	Logic (propositional and predicate logic, resolution algorithms)		
3			
4			Worksheets: 6% (four fortnightly formative tasks on algorithms for propositional logic, regular languages, and formal grammars)
5		Assign. 1: 12% (mathematical and algorithmic logic challenges)	
6	Discrete maths (sets, functions, relations)		
7			
8	Formal languages (DFAs, NFAs, reg. expressions, CFGs, PDAs, Turing machines)	Assign. 2: 12% (challenges in discrete maths and formal languages)	
9			
10			
11			
12	Computability		
Exams		Exam: 70%	

Programming to learn

Grok Academy (web-based programming learning tool) + custom Haskell exercises

- **Description:** traditional logic/TCS exercise (e.g. convert NFA to DFA)
- **Scaffolding:** Haskell type represents formal object (e.g. DFA as 5-tuple)
- **Tests:** custom analysis algorithms (e.g. DFA equivalent to solution)



Many exercises can be framed as instance exercises (define some structured object; integer, formula, DFA, etc.), others as implementation exercises (define a function).

Wait, what about proofs?

We don't want to sacrifice proof exercises (make a formal argument to establish a proposition).

- Proof exercises are effective for testing deep understanding.
- Proof techniques may be considered an important learning goal in their own right.

We found that many of the proofs focus on a simple "constructive algorithm", so we use that for an implementation exercise:

- Students implement the constructive algorithm as a Haskell function.
- This requires the same deep (detailed) understanding of how to analyse and construct formal objects.
- No justification of construction's correctness.
- We can provide an alternative 'check' through automated testing.

Example proof (construction) exercise

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

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

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* $d > 0$*).*

Haskell representation of DFAs:

```
type DFA = ( [Int]   -- states
           , [Char]  -- alphabet
           , [((Int, Char), Int)]
             -- transition relation
           , Int     -- start state
           , [Int]   -- accept states
           )
```

Haskell construction answer:

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

Is functional programming the right tool?

Students are gaining 'hands-on' experience with formal objects and constructions, sometimes with rapid test-based feedback.

But constructions are not complete proofs (missing formal justifications). Consider, say, proof assistants?

Haskell syntax really shines in some examples—very close to mathematical formalism.

On the other hand, sometimes the syntax lets some unnecessary details get in the way.

If students are more comfortable with programming than with mathematics, we think coding with objects can help.

Our students are not fluent with Haskell. New paradigm adds significant overhead. Consider other languages?

An important direction for future work is evaluation, especially from the student perspective.

Questions?

Any questions?

More information:

M. Farrugia-Roberts and H. Søndergaard, "Teaching simple constructive proofs with Haskell programs", extended abstract at TFPIE 2022, full paper in preparation.

M. Farrugia-Roberts, B. Jeffries, and H. Søndergaard, "Programming to learn: Logic and computation from a programming perspective", ITiCSE 2022, to appear.

Grok Academy website: grokacademy.org/universities/

Models of Computation course details: handbook.unimelb.edu.au/subjects/comp30026

Another example (bonus slide)

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

Haskell representation of propositional logic expressions:

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

Haskell construction answer:

```
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))
```

Assessment of construction exercises (bonus slide)

As digital exercises, constructive proof programs are amenable to marking automation.

Our marking work-flow for these questions may look something like this:

- Check that the students function compiles. If not, forgive minor typos, or assess irredeemable functions manually for partial credit.
- If it compiles, run the function on a sample of test inputs.
- A pre-prepared analysis script verifies properties of the outputs.
- If the function passes a large number of tests, consider it correct, and award full marks.
- If the function fails some tests, cluster it with other functions that have the same test behaviour.
- For each cluster, review the behaviour and (optional) the source of each function, and assess manually for partial credit.

We also provide selective, low-level feedback to students during assessment.

More on 'programming to learn' (bonus slide)

Our broader approach has been to use Haskell as a medium for all kinds of exercises.

For example we find that many pen and paper exercises can be programmified into tasks where students use Haskell as an embedded DSL to specify their answer.

We have reflected upon the benefits and costs of this approach in a paper (to appear, ITiCSE 2022):

Logistic benefits

- Rich digital exercise format
- Rapid exercise type development
- Unified interface across topics

Pedagogical benefits

- Rapid formative feedback
- 'Hands-on' engagement
- Student empowerment

We have identified similar barriers to those discussed above.

Once again an important direction for future work is evaluation.

Related work

It is not a completely new idea to use programs to teach proof techniques.

- Lots of work using proof assistants in logic classes

We are impressed by the broad range of topics we can cover with a single programming language. Tools using Haskell appear rarer:

- Leipzig Autotool of Johannes Waldmann:
  www.imn.htwk-leipzig.de/~waldmann/autotool/
  - Includes many instance exercises in logic, automata, discrete math, and more topics.
  - Includes a ‘pumping lemma game’ with functions, similar to (more complex than) our constructive proof exercises.
- Others???