A Course in Haskell-Based Software Testing

Jan van Eijck
Software Analysis and Transformation
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
jve@cwi.nl

Vadim Zaytsev
Informatics Institute
Universiteit van Amsterdam
Amsterdam, The Netherlands
vadim@grammarware.net

1 Software Testing for Software Engineers

The curriculum of the one-year Master of Science Programme in Software Engineering at the University of Amsterdam[1] starts with a course in Haskell-based software testing. Other courses in the curriculum focus on software construction, software evolution, software architecture, software process, and requirements engineering. The goal of the Master of Software Engineering program is to transform bachelors in computer science with reasonable programming experience into full-fledged software engineers within one year. Around 60 students graduate from this program per year.

The functional programming perspective on testing was chosen to make formal methods digestible for students without extensive formal background. We see it as a more apparently practical alternative to other less strict approaches to formal methods like teaching VDM [FF93]. Logic reasoning and mathematical notation are brought in where applicable, but wherever possible Haskell itself is used as a specification language, and type specifications for functional programs are presented as examples of software specification. The motivation communicated to students is to make and test quality software — i.e., software that is maintainable, reliable, efficient and user-friendly [Som10].

The current set-up of the course has the following ingredients:

1. Two hours of lecturing per week.
2. Two hours of workshop sessions per week, in groups of 20 students. The sessions consist of blackboard exercises in formal thinking, with applications to programming.
3. Two days per week of lab work, consisting of programming and testing exercises that are closely linked to the contents of the lectures, with a submission deadline each week.
4. Reading assignments from the book [DvE12], intended to cover Chapters 1 through 7: formal reasoning with propositional and predicate logic, sets and set notation, relations and relational properties, functions as relations, induction and recursion on numbers, lists, trees, and more generally, handling recursive data structures.

The student audience is heterogeneous, and this programme is perceived as very hard by some and relatively easy by others. In the next sections, we will give examples of how we use Haskell to get the importance across of specification as a basis for testing. We will do this by discussing elements of the course, explaining in each particular case what the element is supposed to teach, and how.

[1] http://www.software-engineering-amsterdam.nl

Submitted to:
TFPIE 2014

2 Understanding (Type) Specifications

The course is problem based and student-driven in the sense that each year we measure the actual initial knowledge of students and adjust the material based on it. Workshops group division is also based on estimated levels of knowledge. The course starts with a quiz. One of the quiz questions introduces three software engineers who are discussing a simple grammar:

$$S ::= a \mid aS.$$

Engineer A says: the grammar generates a finite language, for every expression generated by the grammar has finite length. Engineer B says: the grammar generates an infinite language, for the grammar can generate expressions that are infinitely long. Engineer C says: the grammar generates an infinite language, but all of the expressions that are generated by the grammar have finite length. Which of them is right, and why?

Maybe surprisingly, this yields many wrong answers. So consider "Sentences can go on and on and on (and on)*" and compare the following two programs.

```
sentence = "Sentences can go " ++ onAndOn
  where onAndOn = "on and " ++ onAndOn

sentences = "Sentences can go on" : map (++ " and on") sentences
```

What are the types? What is the connection with the dispute between the software engineers? Can you now resolve the dispute?

Next, we show how Haskell can be used to solve the famous Lady or Tiger puzzles of Raymond Smullyan [Smu09]. In the first puzzle, there are two rooms, and a prisoner has to choose between them. Each room contains either a lady or a tiger. In the first test the prisoner has to choose between a door with the sign "In this room there is a lady, and in the other room there is a tiger", and a second door with the sign "In one of these rooms there is a lady and in the other room there is a tiger." A final given is that one of the two signs tells the truth and the other does not.

Towards a Haskell implementation that states the puzzle, we first introduce a data type:

```
data Creature = Lady | Tiger deriving (Eq,Show)
```

Given this, it is clear what the specifications of the types for the signs on the two doors should be:

```
sign1, sign2 :: (Creature,Creature) -> Bool
```

And the messages are as follows:

```
sign1 (this,other) = this == Lady && other == Tiger
sign2 (x,y) = x /= y
```

The first challenge for the students is to find the type of the solution. Now we can explain the concept of a logical space, and the notions of ignorance about the solution and knowledge of the solution.

```
solution :: [(Creature,Creature)]
```

Once the students understand this, it is easy to explain how the solution can be given by applying the constraint "one of the two signs tells the truth and the other does not."

```
solution = [ (x,y) | x <- [Lady,Tiger],
                     y <- [Lady,Tiger],
                     sign1 (x,y) /= sign2 (x,y)]
```

Now, here is the specification of a well-designed Lady and Tiger puzzle: the constraints should be such that there is a *single solution*. This is the recipe for becoming a logic puzzle designer. Without looking at the Smullyan book, design some Lady and Tiger puzzles of your own, and use Haskell to test whether your puzzles are well-designed.

For this, it is useful to introduce a *solve* procedure:

```
solve p = [ (x,y) | x <- [Lady,Tiger],
                    y <- [Lady,Tiger],
                    p (x,y)           ]
```

What is the type of *solve*?

Here is an example puzzle. It is given that either both signs assert something that is true, or both signs assert falsehoods. The first sign says: "In both of these rooms there are ladies." The second sign says: "In this room there is a lady, in the other room there is a tiger." Is this a well-designed puzzle? No, it is not:

```
sign1' (this,other) = this == Lady && other == Lady
sign2' (other,this) = other == Tiger && this == Lady

solution1 = solve (\ (x,y) -> sign1' (x,y) == sign2' (x,y))
```

This gives: `solution1 == [(Lady,Tiger),(Tiger,Tiger)]`. This indicates that we have to add a constraint. The King says to the prisoner: "If you make the right choice, it is sure that you will not get eaten." Does this give us a well-designed puzzle? Yes, it does:

```
solution2 =  solve (\ (x,y) -> sign1' (x,y) == sign2' (x,y)
                            && not ((x,y) == (Tiger,Tiger)))
```

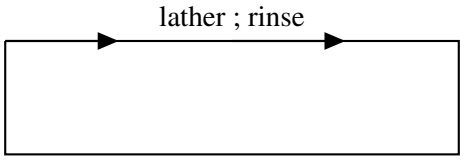
This gives `solution2 == [(Lady,Tiger)]`. The students are now well on their way to become logic puzzle designers, and they have understood what it means for a logic puzzle to meet its specification. This naturally generalises to derivation of any data under a given set of constraints (such as test data generation).

3 Functional Imperative Style

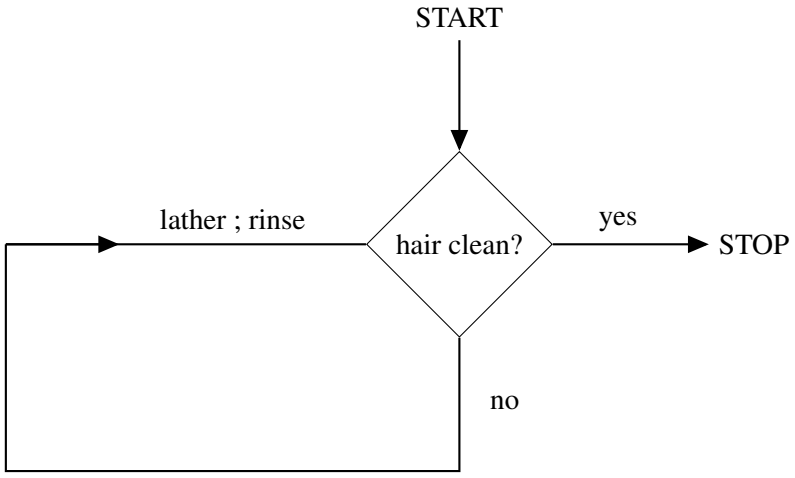
The point we want to get across in the course is that the crucial ingredient of a good testing technique is adequate specification of what your program is supposed to do.

We make short shrift with the prejudice that the functional programming paradigm is far removed from imperative programming. For that, we discuss the structure of a while loop.

If taken literally, the compound action "lather, rinse, repeat" would look like this:



So it makes sense to introduce a stop condition: Repeat the lather rinse sequence until your hair is clean. This gives a more sensible interpretation of the repetition instruction:



So we see that the two ingredients of a while loop are:

- a *test for loop termination*;
- a *step function* that determines the parameters for the next step in the loop.

The termination test takes a number of parameters and returns a boolean, the step function takes the same parameters and computes new values for those parameters.

Here is another example loop:

Integer Decomposition Algorithm

- while even y do
  $x := x + 1$;
  $y := y \div 2$.

The functional version has the loop replaced by a recursive call:

```
g (x,y) = if even y then g (x+1,y `div` 2)
                    else (x,y)
```

But why not introduce an explicit *while* functional?

```
while :: Eq a => (a -> Bool) -> (a -> a) -> a -> a
while p f = \ x -> if p x then while p f (f x)
                          else x
```

Alternative definition of `while` (but a bit harder to read):

```
while = until . (not.)
```

This allows us to write:

```
decomp = while (even.snd) (\ (x,y) -> (x+1,y `div` 2))
```

Euclid's GCD algorithm

1. while $x \neq y$ do
   if $x > y$ then $x := x - y$ else $y := y - x$;
2. return y.

To put Euclid's algorithm in functional imperative style, we introduce a version of *while* with two parameters:

```
while2 :: (a -> b -> Bool)
       -> (a -> b -> (a,b))
       -> a -> b -> b
while2 p f x y
  | p x y     = let (x',y') = f x y in
                            while2 p f x' y'
  | otherwise = y
```

This allows us to write:

```
euclidGCD :: Integer -> Integer -> Integer
euclidGCD = while2
             (\ x y -> x /= y)
             (\ x y -> if x > y
                       then (x-y,y)
                       else (x,y-x))
```

4 Hoare Assertions as Testable Specifications

Some abbreviations for the logic of specifications:

```
infix 1 ==>
p ==> q = (not p) || q

forall = flip all
```

A (Hoare) assertion about an imperative program [Hoa69] has the form

$$\{\text{Pre}\}\ \text{Program}\ \{\text{Post}\}$$

where *Pre* and *Post* are conditions on states.

This Hoare statement is true in state s if truth of *Pre* in s guarantees truth of *Post* in any state s' that is a result state of performing *Program* in state s.

One way to write assertions for functional code is as wrappers around functions. This results in a much stronger sense of self-testing than what is called *self-testing code* (code with built-in tests) in test driven development [Bec02].

The precondition of a function is a condition on its input parameter(s), the postcondition is a condition on its value.

Here is a precondition wrapper for functions with one argument. The wrapper takes a precondition property and a function and produces a new function that behaves as the old one, provided the precondition is satisfied.

```
pre :: (a -> Bool) -> (a -> b) -> a -> b
pre p f x = if p x then f x
            else error "pre"
```

A postcondition wrapper for functions with one argument:

```
post :: (b -> Bool) -> (a -> b) -> a -> b
post p f x = if p (f x) then f x
             else error "post"
```

Example use:

```
decmp = post (odd.snd) decomp
```

More generally, an assertion is a condition that may relate input parameters to the computed value. Here is an assertion wrapper for functions with one argument. The wrapper wraps a binary relation expressing a condition on input and output around a function and produces a new function that behaves as the old one, provided that the relation holds.

```
assert :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert p f x = if p x (f x) then f x
               else error "assert"
```

Example use:

```
decompA = assert (\ (i,j) (m,n) -> 2^i*j == 2^m*n) decomp
```

Let `factors :: Integer -> [Integer]` be a function that computes the list of (prime) factors of an integer. Let's say this function is implemented as follows:

```
factors :: Integer -> [Integer]
factors n = factors' n 2 where
  factors' 1 _ = []
  factors' n m
    | n `mod` m == 0 = m : factors' (n `div` m) m
    | otherwise      =     factors' n (m+1)
```

What would a reasonable assertive version of the `factors` function look like? Answer:

```
factorsA :: Integer -> [Integer]
factorsA = assert (\ x xs -> x == product xs) factors
```

A postcondition wrapper for functions with two arguments:

```
post2 :: (c -> Bool) -> (a -> b ->c) -> a -> b -> c
post2 p f x y = if p (f x y) then f x y
                else error "post2"
```

As an example we specify the GCD of two integers as a postcondition. The definition of GCD is given in terms of the *divides* relation. An integer n divides another integer m if the process of dividing m by n leaves a remainder 0.

```
divides :: Integer -> Integer -> Bool
divides n m = rem m n == 0
```

An integer n is the GCD of k and m if n divides both k and m, and every divisor of k and m also divides n.

```
isGCD :: Integer -> Integer -> Integer -> Bool
isGCD k m n = divides n k && divides n m &&
              forall [1..min k m]
              (\ x -> (divides x k && divides x m)
                        ==> divides x n)
```

This yields the following self-testing version of Euclid's GCD algorithm:

```
euclidGCD' :: Integer -> Integer -> Integer
euclidGCD' k m = post2 (isGCD k m) euclidGCD k m
```

An *invariant* of a program P in a state s is a condition C with the property that if C holds in s then C will also hold in any state that results from execution of P in s. Thus, invariants are Hoare assertions of the form:

$$\{C\}\ \text{Program}\ \{C\}$$

If you wrap an invariant around a step function in a loop, the invariant documents the expected behaviour of the loop.

Assuming the program to be a function of type `a -> a`, the following code wraps an invariant around the program:

```
invar :: (a -> Bool) -> (a -> a) -> a -> a
invar p f x =
  let
    x' = f x
  in
  if p x ==> p x' then x'
  else error "invar"
```

This is used in the following code, which provides a built-in test that decomposition of $(0,n)$ yields a pair (i,j) with the property that j is odd and $n = 2^i \cdot j$.

```
decomp' :: Integer -> (Integer,Integer)
decomp' n = decmp (0,n) where
  decmp = while (\ (_,m) -> even m)
                  (invar
                    (\ (i,j) -> 2^i*j == n)
                    (\ (k,m) -> (k+1,m `div` 2)))
```

The examples make clear that the essence of writing testable code is writing useful specifications, in the form of assertions, preconditions, postconditions and invariants.

More often than not, an assertive version of a function is much less efficient than the regular version: the assertions are inefficient specification algorithms to test the behaviour of efficient functions. However, this hardly matters: to turn assertive code into self-documenting production code, all you have to do is load a module with alternative definitions of the assertion and invariant wrappers.

Take the definition of `assert`. This is replaced by:

```
assert :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert _ = id
```

And so on for the other wrappers. So the self-testing specifications do not burden the code, for in the production version we can redefine the specifications.

Suppose a program (implemented function) fails its implemented assertion. What should we conclude? This is a pertinent question, for the assertion itself is a piece of code too, in the same programming language as the function that we want to test. So what are we testing:

- the correctness of the code?
- the correctness of the implememented specification for the code?

In fact, we are testing both at the same time. Therefore, the failure of a test can mean several things, and we should be careful to find out wat our situation is:

1. There is something wrong with the program.
2. There is something wrong with the specification of the assertion for the program.
3. There is something wrong with the program *and* with its specification.

It is up to us to find out which case we are in. In any case it is important to find out where the problem resides. In the first case, we have to fix a code defect, and we are in a good position to do so because we have the specification as a yardstick. In the second case and third case, we are not ready to fix code defects. First and foremost, we have to fix a defect in our understanding of what our program is supposed to do. Without that growth in understanding, it will be very hard indeed to detect and fix possible defects in the code itself.

5 Further Topics

Sudoku problems have a straightforward specification. If a declarative specification is to be taken seriously, all there is to solving sudokus is *specifying what a sudoku problem is*, and developing the code from this. That is what we do in the course. Next, we develop a sudoku problem generator that can be used to test the solver.

Another advanced topic is modular arithmetic for public key cryptography. We implement and test various (probabilistic) algorithms for fast prime recognition, starting from the naive Fermat method, and ending with the Miller-Rabin algorithm [Mil76, Rab80]. To test these, we use generators for composite numbers, and for Carmichael numbers (odd composite numbers n that satisfy Fermat's little theorem $a^{n-1} \equiv 1 \bmod n$ for any a with $\gcd(a,n) = 1$). This gives a chance to explain the basics of public key cryptography, explain Diffie-Hellman key exchange [DH76], and ultimately explain, implement and test RSA encryption and decryption [RSA78].

Further topics are stable matching algorithms and graph algorithms. For stable matching, the key is to understand the notion of stability, and to recognise that the following function implements the specification.

```
isStable :: Wpref -> Mpref -> Engaged -> Bool
isStable wf mf engaged =
    forall engaged (\ (w,m) -> forall engaged
          (\ (w',m') -> (wf w m' m ==> mf m' w' w)
                         &&
                        (mf m w' w ==> wf w' m' m)))
```

The students are expected to recognise that this implements the famous definition of [GS62].

$$\forall (w,m) \in E \; \forall (w',m') \in E \qquad ((\mathrm{pr}_w m'm \to \mathrm{pr}_{m'} w'w) \wedge (\mathrm{pr}_m w'w \to \mathrm{pr}_{w'} m'm)).$$

Understanding this definition is the key to understanding construction of stable matchings. After this, we discuss, implement and test algorithms for stable matching and college admission.

We also discuss, implement and test algorithms for transitive closure in simple graphs and for shortest path in weighted graphs. The main difficulty here is to explain to the students that exercises about relational structures like graphs provide an abstract perspective on problems that going to occur often in their actual practice as software engineers. We have learnt from experience that it is very important to give many applications, and point out where the formal techniques from logic and algorithm design meet the practical world of software engineering. Once the students learn to recognise binary relations they start to see that transitive closures are everywhere. Then they can answer questions like the following, and what is more, they can appreciate that the answers are relevant for software engineering.

Let R be a binary relation on A. Two elements x and y of A are called *weakly R-connected* if there is a path of forward or backward R steps from x to y. It is allowed that this path is empty, so every point is weakly R connected to itself.

Let `Rel a` be the type `[(a,a)]`. Suppose a function

```
tc :: Ord a => Rel a -> Rel a
```

for the transitive closure of a relation and a function

```
inv :: Ord a => Rel a -> Rel a
```

for inverting a relation are given. Use these to define a function

`wConnected :: Ord a => Rel a -> a -> a -> Bool`

and indicate how you would argue or test that your implementation is correct. Answer:

```
wConnected :: Ord a => Rel a -> a -> a -> Bool
wConnected r x y =
   x == y || elem (x,y) (tc (r ++ (inv r)))
```

Obviously, from any point x there is a path from x to itself. A non-empty path from x to y exists iff x and y are connected by the transitive closure of $R \cup R^{-1}$, and that is precisely what the implementation says.

Let R be a binary relation on A. R is called cycle-free if for no x in A it is the case that there is a non-empty path of forward or backward R steps from x to x. Write a property for this (hint: use your answer to the previous question, or at least your method to answer that question):

```
cycleFree :: Ord a => Rel a -> Bool
cycleFree r = ...
```

Fill in the dots. Next, indicate how you would argue for the correctness of your implementation. Answer:

```
cycleFree :: Ord a => Rel a -> Bool
cycleFree r = let
    s = tc (r ++ (inv r))
  in
    all (\ (x,y) -> x /= y) s
```

In a cycle-free relation all non-empty paths consisting of forward and backward R step are forbidden, so what we have to do is check their existence for any pair of nodes x and y. That is precisely what the implementation does.

The students are led to learn how see the correspondence between `tc (r ++ (inv r))` and $(R \cup R^{-1})^+$, and how to link abstract relations R to the kinds of relations that they encounter in real life, such as the procedure call relations in software modules, the path relations in directory structures, the hyperlink relations on internet, and so on. Their skills of manipulating relations are subsequently reused and strengthened in the immediately following *Software Evolution* course where another functional language called Rascal [KvdSV11] is used to complete metaprogramming tasks such as automated analysis and visualisation of large software systems.

6 Constructive Alignment

Formally, the learning objectives of the course, in Bloom's (revised) terms [AKA+00], are:

- to recognise various testing techniques applied in practical software engineering;
- to compare testing techniques by applicability in certain scenarios;
- to implement formal specifications of software systems and test such systems for conformance;
- to differentiate among alternative approaches to testing and argue in favour of one against another;
- to judge efficiency of a given model for testing.

It is obvious that these are too difficult to reach with the lectures and practical assignments alone, which is why another simultaneously running course (*Academic Skills*) is synchronised with the *Software Testing* course and trains students in locating, consuming, assessing and combining scientific publications. In particular, in 2013 they were confronted at the very start with a large TOSEM paper about evaluating testing techniques [SM12]. All students appreciated that the paper was recent, but it took considerable effort to process it, both for recent Bachelor students and for those with primarily industrial experience. At the end of the two months, their academic skills were at a level where they all demonstrated their ability to write an report logically reasoning for a chosen standpoint by reusing independently sought journal and conference papers.

The actual depth of the material we could cover depended heavily on the factual starting expertise of the students and was adjusted accordingly. Thus, a lot of time was spent during lectures and workshops on rehashing the foundations of logic and formal methods, based on the first chapters of *The Haskell Road* [DvE12]. On the other hand, the knowledge of basic practical software engineering skills related to testing (such as JUnit or Jenkins) was not missed because the students mostly have already come in contact with such technologies prior to our course. The teaching load was unsurprisingly at its highest points for giving detailed feedback on students' source code (to help changing the style, per section 3) and students' essays (mostly to prepare them for the imminent final graduation project).

The final grade in the course was determined by a final theoretical written examination (30%), a final practical examination in Haskell programming (30%) and a cumulative score of the assignments completed by a group during the semester (40%). The course was positively evaluated by the students (an average score for each question at least at 3 out of 5), yet with a disturbing number of remarks stating that they still do not see how specifications relate to the practice of software engineering. This supports our claim of impossibility of teaching them more mainstream or more advanced formal methods techniques like VDM [FF93] to some extent, but also shows that there is space for future improvement.

In several years of experimenting with the course, there is a movement away from emphasis on the lectures to emphasis on problem-related feedback and instruction. We also count not help noticing particular similarities between our course set-up and the flipped classroom [Kin93, LPT00] paradigm:

- many students, especially those initially weaker in functional programming, took complementary reading material very seriously and in the end regretted that it was not always the case that a lecture was linked to particular chapters of the book;
- most time spent in classroom was dedicated, both by design and de facto, to solving practical assignments and acquiring programming and specification skills, not on listening to explanations of the underlying theory;

- two most appreciated components of the course by the students were workshops where they got direct feedback and were solving assignments in direct and close collaboration with the teachers, and detailed feedback on their Haskell code;
- we put some relatively advanced technology to use: each student group had its own git repository for version control of their source code — the teachers had access to all repositories and were able to leave feedback remotely by commenting on commits, making pull requests, reporting issues, etc;
- some kind of ad hoc learning analytics was used to resolve inter-group problems, and individual grades within a group in several cases needed to be adjusted based on commits made by each person;
- most problems we experienced were related either to heterogeneous backgrounds of students (ranging from complete novices in functional programming to having had strong Haskell courses in the past) or to inadequacy in meeting their expectations by insufficiently demonstrating the links existing between the theoretical part (mathematical induction, equivalence classes, lambda calculus, type systems) and the everyday practice of a future software engineering practitioner.

For the future, we are planning to introduce the following components to the course:

- guest lectures on the use of specification-based software testing in practice, since we have observed ourselves their good impact in other courses of our programme;
- total flip of the classroom with pre-recorded lectures and open discussions in lecture time, to continue the already established evolution trajectory for this course;
- peer assessment instead of or in addition to teacher feedback, since there seems to be strong positive evidence for groups of similar size and level [VHG14];
- automated tutoring as immediate supplement to teacher feedback on the source code, which also seems to work well for courses in other universities[2] [HJ14];
- demanding upfront knowledge of functional programming, even if from a non-Haskell or pre-Haskell source [BW88].

In general, we hope to make better utilisation of teachers by spending time in class discussing the questions actually raised after studying the material; we hope to provide means to students to pace their own learning process and thus ensure some minimum learnt by everyone disregarding their initial level; and we certainly hope to collect more data from diagnostics and analytics in order to keep improving the course in future years. Given that Master students usually do not suffer from the lack of motivation (which is commonly presented as one of the biggest downsides of the flip [Ash12]) and that all these elements were met with enthusiasm whenever experienced, we have high expectations for the future of this course.

[2] Cf. Ask-Elle: `http://ideas.cs.uu.nl/FPTutor/`.

References

[AKA+00] Lorin W. Anderson, David R. Krathwohl, Peter W. Airasian, Kathleen A. Cruikshank, Richard E. Mayer, Paul R. Pintrich, James Raths, and Merlin C. Wittrock. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Allyn & Bacon, 2000.

[Ash12] Katie Ash. Educators View "Flipped" Model With a More Critical Eye. *Education Week*, 32(2), August 2012.

[Bec02] Kent Beck. *Test Driven Development By Example*. Addison-Wesley Longman, Boston, MA, 2002.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[DH76] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DvE12] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming, Second Edition*, volume 4 of *Texts in Computing*. College Publications, London, 2012. First Edition: 2004.

[FF93] Neville J. Ford and Judith M. Ford. *Introducing Formal Methods — a Less Mathematical Approach*. Ellis Horwood series in computers and their applications. Ellis Horwood, 1993.

[GS62] D. Gale and L. Shapley. College Admissions and the Stability of Marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[HJ14] Bastiaan Heeren and Johan Jeuring. Feedback Services for Stepwise Exercises. *Science of Computer Programming*, 88:110–129, 2014.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):567–580, 583, 1969.

[Kin93] Alison King. From Sage on the Stage to Guide on the Side. *College Teaching*, 41(1):30–35, 1993.

[KvdSV11] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In João Miguel Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *LNCS*, pages 222–289. Springer, January 2011.

[LPT00] Maureen J. Lage, Glenn J. Platt, and Michael Treglia. Inverting the Classroom: A Gateway to Creating an Inclusive Learning Environment. *The Journal of Economic Education*, 31(1):30–43, 2000.

[Mil76] Gary L. Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[Rab80] Michael O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[SM12] Jaymie Strecker and Atif M. Memon. Accounting for Defect Characteristics in Evaluations of Testing Techniques. *ACM Transactions on Software Engineering and Methodology*, 21(3):17:1–17:43, 2012.

[Smu09] Raymond M. Smullyan. *The Lady or the Tiger? and Other Logic Puzzles*. Dover, 2009. First edition: 1982.

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2010.

[VHG14] Andrii Vozniuk, Adrian Holzer, and Denis Gillet. Peer Assessment Based on Ratings in a Social Media Course. In Matthew D. Pistilli, James Willis, Drew Koch, Kimberly E. Arnold, Stephanie D. Teasley, and Abelardo Pardo, editors, *Learning Analytics and Knowledge Conference (LAK '14)*, pages 133–137. ACM, 2014.

Appendix: Relations Library

```
type Rel a = [(a,a)]

infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]

lfp :: Ord a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)

tc :: Ord a => Rel a -> Rel a
tc r = lfp (\ s -> (sort.nub) (s ++ (s @@ s))) r

inv :: Ord a => Rel a -> Rel a
inv = map (\ (x,y) -> (y,x))

nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)

sort [] = []
sort [x] = [x]
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys) | x > y     = y : (insert x ys)
                | otherwise = x : y : ys
```