Model solutions and properties for diagnosing student programs in Ask-Elle

Johan Jeuring

Thomas van Binsbergen

Alex Gerdes

Bastiaan Heeren

Technical Report UU-CS-2014-025
September 2014

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Model solutions and properties for diagnosing student programs in Ask-Elle

JOHAN JEURING
Utrecht University and Open University of the Netherlands, Heerlen, The Netherlands
and
L. THOMAS VAN BINSBERGEN
Utrecht University, The Netherlands
and
ALEX GERDES
QuviQ, Sweden
and
BASTIAAN HEEREN
Open University of the Netherlands, Heerlen, The Netherlands

Ask-Elle is an interactive tutor that supports the stepwise development of simple functional programs. Using Ask-Elle students receive feedback about whether or not they are on the right track, they can ask for a hint when they are stuck, and get suggestions about how to refactor their program. Our tutor generates this feedback from model solutions and properties that a solution should satisfy. This paper studies the feasibility of using model solutions together with the desired properties of solutions to analyse the work of a student. It describes an experiment in which we analyse almost 3500 log entries from students using Ask-Elle to solve functional programming exercises, to determine how many of these programs are diagnosed correctly based on model solutions and the desired properties of solutions. Ask-Elle manages to correctly diagnose 82.9% of the student programs. A further analysis of the student programs and the diagnoses shows that adding some reasonable model solutions, properties of model solutions, and general program transformations would increase this percentage to 92.9%.

Categories and Subject Descriptors: K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI); K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms: Languages, Human Factors, Measurement

Additional Key Words and Phrases: Functional programming, Haskell, tutoring

ACM Reference Format:

Johan Jeuring, L. Thomas van Binsbergen, Alex Gerdes, and Bastiaan Heeren. 2014. Model solutions and properties for diagnosing student programs in Ask-Elle. Submitted to Computer Science Education Research Conference (CSERC'14).

1. INTRODUCTION

We have developed Ask-Elle [Jeuring et al. 2011; Gerdes et al. 2012], an interactive tutor that supports the stepwise development of simple functional programs in the lazy, pure, higher-order functional programming language Haskell [Peyton Jones et al. 2003]: see Figure 1 for a screenshot. Using this tutor, students learning functional programming develop their programs incrementally, receive feedback about whether or not they are on the right track, can ask for a hint when they are stuck, and get suggestions about how to refactor their program. The order in which a student constructs a program using our tutor is quite flexible. The interactive tutor can diagnose a student step, give hints at each step, generate worked-out solutions for exercises, and recognise common errors made by students. All of this functionality is calculated automatically from the teacher-specified annotated solutions ('model' solutions) for an exercise. This allows a teacher to use her favourite exercises. The tutor is offered as a web application.[1] The unique features of Ask-Elle are that it is sufficient to specify model solutions to add an exercise to the tutor, and the feedback is not only given on final programs, but also on incomplete programs that still need to be completed. Intelligent tutors that support a student step-wise solving a task, by diagnosing partial student solutions or giving hints, are almost as effective as human tutors [VanLehn 2011].

To analyse the work of a student, we compare possibly partial student programs against model solutions. If a student program can be matched against a model solution, Ask-Elle approves it; if it cannot be matched, Ask-Elle asks the student to submit another

Address: Dept. of Information and Computing Sciences, Utrecht University P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands.
E-mail: J.T.Jeuring@uu.nl, ltvanbinsbergen@acm.org, alex.gerdes@quviq.com, bastiaan.heeren@ou.nl.

DOI: http://dx.doi.org/10.1145/XXXXXXX.YYYYYYY

[1] http://ideas.cs.uu.nl/FPTutor/. Use any name to log in.

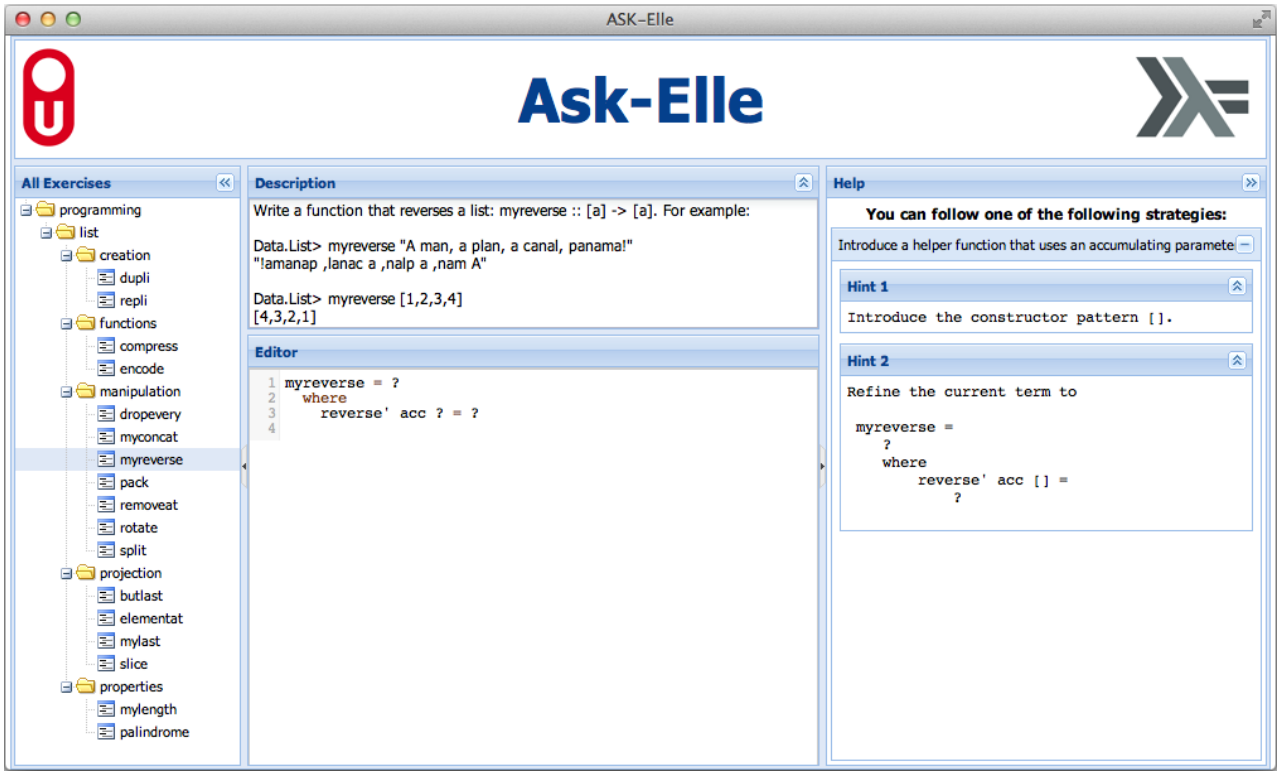

Fig. 1. Screenshot of Ask-Elle

program. We want the analysis of Ask-Elle to be as precise as possible, so we want to approve as many as possible correct student programs. If we do not approve of a student program, we would want it to be indeed incorrect. Assume we have some means to determine exactly whether or not a program is (in)correct. We call this means an oracle. The exact form of the oracle is not important, it might consist of a panel of teachers, or some advanced algorithm that can determine the correctness of programs. We would like the generated feedback to be *sound* and *complete* with respect to the oracle. Generated feedback for a student program is sound when a student program is diagnosed to be (in)correct, it is (in)correct according to the oracle, i.e. no false positives or false negatives are generated. Generated feedback is complete when the oracle diagnoses a student program to be (in)correct, the software also diagnoses it to be (in)correct, i.e., our tutor should always be able to give a definitive diagnosis.

To approve as many correct student programs as possible, we apply several transformations on both the student program and the model solution. For example, we apply desugaring, alpha-conversion, and beta-reduction. Desugaring removes syntactic sugar from a program, such as infix operators (for example, $x + 2$ is transformed into $(+)\ x\ 2$) and pattern matching (transformed to a **case** expression). Alpha-conversion renames the bound variables in a program in a uniform way. Beta-reduction removes redeces in a program: if we can apply a reduction, we do so (for example, $\mathbf{if}\ p \vee \mathit{True}\ \mathbf{then}\ q\ \mathbf{else}\ r$ is replaced by q). If we cannot match a student program against a model solution, we test the student program against properties that the solution should satisfy, to find out if possibly the student program is incorrect. We use QuickCheck [Claessen and Hughes 2000] to specify properties of solutions, and to search for counterexamples. If QuickCheck finds a counterexample, we report it to the student. Thus we use both static (matching against model solutions) and dynamic (testing against properties) techniques to analyse student programs.

Semantic equality of programs is undecidable, and there does not exist a normal form for programs. This implies that we cannot give a complete set of transformations to use when transforming student programs and model solutions, and hence that we sometimes cannot automatically match a student program against a model solution. In such cases, we revert to QuickCheck to test whether or not a student program is behaving the same as a model solution, but QuickCheck cannot always give a definitive answer. This leaves us once in a while in the undesirable situation where we cannot say anything about the student program. We don't know if the program is correct or incorrect. There are two causes of this problem:

(1) Not matching any of the model solutions:
—The student has implemented a new solution to the exercise. In this case we should add that solution to our set of model solutions.
—Our set of program transformations is too limited to transform a student program to a model solution, in which case we should extend the set of program transformations.

(2) QuickCheck gave up:
—If many parts of a student program are not yet defined, QuickCheck may not be able to test the stated properties.
—A property may have too strong a precondition or a poor generator for a particular datatype. In this situation QuickCheck discards too many test cases.

We have performed an experiment with Ask-Elle at Utrecht University with second-year students taking a course on functional programming, to find out if there are many cases in which Ask-Elle cannot determine whether a student program is correct or incorrect, and what the cause of this is. We analysed almost 3500 log entries from students using Ask-Elle to find entries which help in finding preliminary answers to questions such as: How many different student programs can we relate to a model solution? What program transformations do we need to transform student programs and model solutions into a normal form, which we use to match programs? What is the percentage of correct student programs not recognised by our tutor? What is the percentage of incorrect programs for which we cannot find a counterexample? How can we reduce these percentages?

This paper makes the following contributions:

—It describes an empirical study in which we analyse student-input to derive statistical information about the feasibility of using model solutions for analysing student programs.

—It shows how model solutions can be extended with *properties* to be used for automated testing.

—It collects information about if and which program transformations are necessary for transforming a student program to a model solution.

If we find that using model solutions together with properties of these solutions is indeed a feasible approach to analysing student programs, we have a strong argument for further developing Ask-Elle, and for using the same approach to develop intelligent tutors for other programming languages or paradigms.

Besides the above contributions, this deep analysis of student submissions revealed several inconsistencies in our model solutions, and led to discussions about the desirability of some of the program transformations, model solutions, and teacher annotations.

Related work. Ala-Mutka [Ala-Mutka 2005] gives a general overview of automatic assessment of programming assignments. Most, if not all, approaches to automatic assessment deal with complete programs, and do not consider the kind of intermediate programs that we also want to diagnose. Our functional programming tutor grew out of a program assessment tool, which automatically assesses complete student programs based on model solutions [Gerdes et al. 2010] and program transformations to rewrite programs to normal form. Similar transformations have been developed for imperative languages. For example, Xu and Chee [Xu and Chee 2003] develop program transformations for C++ to transform complete student programs to model solutions. Singh et al. [Singh et al. 2013] use program transformations to rewrite erroneous imperative programs into model solutions, generating feedback on the way. Jin et al. [Jin et al. 2012] generate feedback based on a graph-based view of imperative programs, using knowledge from a database of student solutions. Rivers and Koedinger [Rivers and Koedinger 2014] also use previously submitted student programs to generate hints in an intelligent tutoring system for (imperative) programming. Using previously submitted student programs is probably a bit easier than implementing the program transformations we envisage, but brings with it the risk that suboptimal solutions appear in the hints.

Organisation of this paper. This paper is organised as follows. Section 2 gives some examples of model solutions. Section 3 shows how we extend Ask-Elle with the possibility to test properties of student programs. Section 4 identifies the situations in which Ask-Elle cannot analyse a student program, and Section 5 analyses a large set of student programs to determine the cause of the problem. Related and future work are given in Section 6, and Section 7 concludes.

2. MODEL SOLUTIONS

Ask-Elle generates feedback by comparing student programs to a set of model solutions. A model solution is an annotated teacher-specified program that makes use of good programming practices. A teacher can specify more detailed feedback by annotating a model solution. This section gives two examples of exercises offered by Ask-Elle, and some of the model solutions for these exercises. Most exercises offered by Ask-Elle come from the Ninety-nine Haskell Problems.[2] Ask-Elle is targeted at students beginning to learn Haskell, and the kind of problems offered require solutions no longer than a couple of lines. These are the kind of programs developed by beginning Haskell programmers.

2.1 Reverse

The $reverse$ exercise asks a student to define a function that reverses a list, which has the following type signature:

$$myreverse :: [a] \to [a]$$

For example:

```
> myreverse "A man, a plan, a canal, panama!"
"!amanap ,lanac a ,nalp a ,nam A"

> myreverse [1,2,3,4]
[4,3,2,1]
```

For $myreverse$, Ask-Elle has three model solutions, among which the standard solution using an accumulating parameter:

$$\begin{aligned} &myreverse = reverse'\ [\,] \\ &\quad \textbf{where} \\ &\qquad reverse'\ acc\ [\,] \quad\ = acc \\ &\qquad reverse'\ acc\ (x : xs) = reverse'\ (x : acc)\ xs \end{aligned}$$

2.2 Rotate

The $rotate$ exercise asks a student to define a function that rotates a list n places to the left:

$$rotate :: [a] \to Int \to [a]$$

For example:

```
> rotate [1,2,3,4,5] 2
[3,4,5,1,2]
```

Ask-Elle has three model solutions for this exercise, among which:

$$\begin{aligned} &rotate\ xs\ n = drop\ i\ xs +\!\!+ take\ i\ xs \\ &\quad \textbf{where}\ i = n\ `mod`\ (length\ xs) \end{aligned}$$

where $drop\ \ n\ \ xs$ drops the first n elements of list xs, and $take\ \ n\ \ xs$ returns the first n elements of xs. Functions $length$ and mod are defined in the Haskell prelude.[3]

[2] http://www.haskell.org/haskellwiki/99_Haskell_exercises

[3] The prelude is the standard library for Haskell containing many useful functions.

3. PROPERTIES OF PROGRAMS

Until recently, if Ask-Elle could not determine whether or not a student program matches a model solution, it could only report that the student program doesn't follow the structure of one of the model solutions. To also report erroneous programs (i.e., programs with the wrong input-output behaviour), we use QuickCheck [Claessen and Hughes 2000] for specifying properties, testing these properties using randomly generated input, and finding counterexamples. Furthermore, each of the properties for which we test is accompanied by a specific error message that explains the kind of error detected. A student can also submit an incomplete program; we use therefore a recent version of QuickCheck, which can ignore undefined parts of a student program.

We could also have used unit testing to check a student program. In our case, property-based testing has the advantage that it randomly generates input values and discards those which the student program cannot yet handle. Thus, even if a student program is defined for only a small subdomain of the target domain, it can still be checked for counterexamples.

Each programming exercise is accompanied by one or more properties that have to be satisfied by a student program for the exercise. A property should hold for *all* solutions to the programming exercise. A student program is only tested if Ask-Elle cannot determine that a student program follows a model solution.

3.1 Properties for $myreverse$

The function $myreverse$ has a number of properties. The first property states that all elements in the input list are also present after reversing:

$$propElems\ xs = xs \subseteq myreverse\ xs$$

Although the operator $\subseteq$ is not present in Haskell's prelude, we can conveniently define it in terms of list difference ($\backslash\backslash$):

$$xs \subseteq ys = null\ (xs \backslash\backslash ys)$$

With this property we can identify errors in student programs. For example, if a student defines $myreverse$ as

$$\begin{aligned} myreverse\ [\,] &= [\,] \\ myreverse\ (x : xs) &= myreverse\ xs \end{aligned}$$

then QuickCheck finds a counterexample such as $[1]$ when it validates the property $propElems$. Presenting this counterexample to a student is helpful, but it might not give her sufficient information about what is wrong in the program. Together with a counterexample, we can report a specific message explaining why the definition is not correct. For example, we can report: *"Some elements in the input do not appear in the output"*. To support the possibility of providing feedback for a specific error, a teacher can adapt or introduce a property for a programming exercise. We add a message to the above property:

$$\begin{aligned} &propElems\ xs = whenFail\ msg\ (xs \subseteq myreverse\ xs) \\ &\quad \textbf{where} \\ &\qquad msg = putStrLn\ (\texttt{"Not all elements in input "} \\ &\qquad\qquad +\!\!+ show\ xs +\!\!+ \texttt{" appear in output "} \\ &\qquad\qquad +\!\!+ show\ (myreverse\ xs)) \end{aligned}$$

The explanatory feedback message is shown whenever a student program does not satisfy the stated property.

It is possible to add multiple properties with specialised messages to an exercise. For example, we can test that no new elements are introduced by $myreverse$:

$$propElems2\ xs = myreverse\ xs \subseteq xs$$

Properties $propElems$ and $propElems2$ together express that function $myreverse$ only permutes the elements of a list. A third property states that $myreverse$ is its own inverse:

$$propTwice\ xs = myreverse\ (myreverse\ xs) == xs$$

It could be tempting to start including general properties that completely specify how $myreverse$ should behave, including how to reverse a list that is split into two parts:

$$\begin{aligned} &propSplit\ xs\ ys = \\ &\quad myreverse\ (xs +\!\!+ ys) == myreverse\ ys +\!\!+ myreverse\ xs \end{aligned}$$

However, we always specify a catch-all property that compares the student program with a trusted implementation. We use either one of the teacher-specified model solutions or a reference implementation (e.g. from the prelude):

$$propReverse\ xs = myreverse\ xs == reverse\ xs$$

When we start testing we first use the more specific properties (such as $propElems$) because these result in the more precise error messages. Testing against a reference implementation is done last. The order in which the properties are specified is also the order in which they are tested.

3.2 Properties for $rotate$

As a second example we give a number of properties for $rotate$. Before we do so we first show two student programs that contain common errors. We illustrate how a teacher could anticipate these mistakes and define properties with specific feedback.

A student might try to solve the $rotate$ exercise in terms of the functions $drop$ and $take$:

$$\begin{aligned} &rotate_s :: [a] \to Int \to [a] \\ &rotate_s\ xs\ n = drop\ n\ xs +\!\!+ take\ n\ xs \end{aligned}$$

This implementation gives the correct results for the example given earlier. However, it assumes that $0 \leqslant n \leqslant length\ xs$, while the exercise specifies no constraints on n. Hence we find a counterexample:

$$\begin{aligned} &> rotate_s\ [1,2,3,4,5]\ 6 \\ &[1,2,3,4,5] \end{aligned}$$

The model solution returns $[2,3,4,5,1]$. Confronted with this example, a student might change the program to:

$$\begin{aligned} &rotate_s :: [a] \to Int \to [a] \\ &rotate_s\ xs\ n \mid n > length\ xs = drop\ i\ \ xs +\!\!+ take\ i\ \ xs \\ &\qquad\qquad\quad \mid otherwise\ \ \ = drop\ n\ xs +\!\!+ take\ n\ xs \\ &\quad \textbf{where}\ i = n\ `mod`\ length\ xs \end{aligned}$$

This definition uses case-distinction to cover the case where n is larger than the input list.

The model solution for $rotate$, given in the previous section, not only takes into account the special case of a full rotation of the list, but also the case of rotating in the opposite direction ($n < 0$). Therefore, QuickCheck again finds a counterexample:

$$\begin{aligned} &> rotate_s\ [1,2,3,4,5]\ (-1) \\ &[1,2,3,4,5] \end{aligned}$$

The expected result is $[5,1,2,3,4]$ while the given definition returns the input unchanged, since $drop$ returns the input list unchanged for negative numbers (and $take$ returns $[\,]$).

For $rotate$, we have formulated a number of general properties. The first property is defined as follows:

$$propAdd\ xs\ n\ m = \\ \quad rotate\ xs\ (n + m) == rotate\ (rotate\ n\ xs)\ m$$

The second general property expresses that rotating by the length of the input list returns the same list, which means that rotating over n is the same as rotating over $n + length\ xs$. Using QuickCheck, this property is expressed as:

$$propWrap\ xs\ n = rotate\ xs\ (n + length\ xs) == rotate\ xs\ n$$

These general properties may not give specific enough feedback for a student to discover what is wrong. We therefore specialise the general properties for the identified corner cases. For large numbers we use the property:

$$propBigN\ xs\ n = n > length\ xs \Longrightarrow propWrap\ xs\ n$$

The infix function ($\Longrightarrow$) is implication for properties. The property is only considered if the precondition holds. The general property $propWrap$ is therefore only validated for n larger than the length of xs.

To cover the corner case for rotating in the opposite direction, we do not need to write a specialised property. Since we first consider the specialised property $propBigN$, we can use $propWrap$ because it considers all integer values for n, including negative ones. To find possible remaining errors we finally validate the catch-all property that compares a student implementation to a model solution.

3.3 Dealing with incomplete programs

An interesting aspect of Ask-Elle is that it can also analyse incomplete programs and determine whether or not a student is on the right track towards a model solution. Programs may contain one or more *holes*, denoted by a question mark, which represent components that a student wants to fill out later. Testing incomplete programs leads to some challenges.

We use QuickCheck's *discard* function, which discards the current test case, to handle an incomplete program. We replace every hole in a program by *discard*. As a result, many test cases will be discarded by QuickCheck. However, even for incomplete programs, some test cases may go through because no hole needs to be evaluated. For example, consider the following implementation of $rotate$:

$$rotate_s :: [a] \to Int \to [a] \\ rotate_s\ xs\ n \mid n < 0 = drop\ ?\ xs +\!\!+ take\ ?\ xs \\ \qquad\qquad\quad \mid n \geqslant 0 = drop\ n\ xs +\!\!+ take\ n\ xs$$

Here QuickCheck finds the counterexample related to the assumption that $n \leqslant length\ xs$:

$$> rotate_s\ [1, 2]\ 3 \\ [1, 2]$$

4. CLASSIFYING STUDENT PROGRAMS

In this section we classify student programs. A full program is classified as correct if it has the expected input-output behaviour. A partial program (with holes) is considered to be correct if replacing the holes with expressions can lead to a correct program. We use the following categories for classifying submitted student programs:

—*Compiler error (Error).* Ask-Elle uses Helium [Heeren et al. 2003] and GHC[4] to compile student programs. Helium has been developed to report good error messages, and GHC is needed to run QuickCheck tests. Both compilers might report a syntax or type error, which the student has to repair.

—*Matches model solution (Model).* Ask-Elle makes use of program transformations to match a student program with a model solution. The student is on the right track solving the exercise, or finished with the exercise (if there are no more holes).

—*Counterexample (Counter).* Based on one of the properties QuickCheck finds a counterexample and reports a specialised message explaining to the student why her program is incorrect.

—*Undecided.* Programs that cannot be matched with a model solution, and without a counterexample, cannot be diagnosed as correct or incorrect by Ask-Elle. Later we will separate this category into *Tests passed*, for programs for which all tests pass, and *Discarded*, for programs for which most test cases are discarded, in almost all cases because the program is still undefined at many places.

Ideally, the number of programs in this last category is small. Programs for which correctness is undecided raise some interesting questions related to the quality of feedback reported by the tutor:

—How many programs are classified as undecided?

—How often would adding a program transformation help with matching against model solutions?

—How often would adding a model solution help?

—How often do students add irrelevant, with respect to the exercise, parts to a program with the correct input-output behaviour?

—How many of the programs with correct input-output behaviour contain imperfections, such as redundant case-clauses, which are perhaps impossible to remove automatically.

—How often does QuickCheck not find a counterexample, although the student program is incorrect?

In the following subsections we take a closer look at why correct programs cannot always be matched with a model solution, and why QuickCheck sometimes cannot find counterexamples for incorrect programs. We give answers to the above questions at the end of Section 5.

4.1 Correct (but no match)

The student program is correct. It is not matched against one of the model solutions because:

(1) The student has come up with a way to solve the exercise that significantly differs from the model solutions.

(2) Ask-Elle misses some transformations to transform the student program and a model solution into the same program.

(3) The student has solved more than just the programming exercise. For example, she has added checks on the input, or elaborate error messages.

(4) The student implementation does not use good programming practices or contains imperfections.

Case (1) leads to adding the student solution as a new model solution to Ask-Elle. This of course raises the question when a solution is a new model solution, and when can a solution be transformed into an existing model solution. In general, it is impossible

[4]Glasgow Haskell Compiler the default compiler for Haskell

to develop a transformation system that can transform any two semantically equal programs into each other [Voeten 2001]. Our basic approach in Ask-Elle has been to only add transformations to Ask-Elle about which we never want to give feedback to students. Existing transformations are mainly related to style issues: the use of names, explicitly specifying arguments, using local definitions, etc. This implies we don't check such style issues in Ask-Elle, although we might use a tool such as HLint[5] for this purpose. In case (2) we should add the transformation to Ask-Elle or improve existing transformations. In case (3) we probably want to report the fact that the student has done too much, provided this can be recognised. Finally, the solutions in case (4) can be regarded as residuals about which Ask-Elle cannot give a precise judgement.

4.2 Incorrect (but no counterexample)

QuickCheck will not always be able to report a counterexample for incorrect programs. Besides finding a counterexample, the outcome of checking the properties can be:

—*Tests passed.* All test cases passed. By default, 100 test cases are run with random values for each property.

—*Discarded.* Too many test cases are discarded. By default, more than 90% is considered to be too many.

In case *Tests passed*, full programs that pass all test cases are likely to be correct; it is very unlikely that programs with incorrect input-output behaviour pass all properties without finding a counterexample. For partial programs (with holes) we have to be a bit more careful since test cases that run into holes are discarded, and this may influence the distribution of random values that are considered. Case *Discarded* is a clear indication that the program is not yet defined enough. Whenever a hole is encountered during evaluation, the test case will be discarded. The outcome is *Discarded* if less than 10% of the test cases can be used. In this case, the other at least 90% of the test cases need parts of the program that have not been defined yet.

5. STUDENT PROGRAM ANALYSIS

We have analysed the log files of Ask-Elle, with 5950 log entries from students attending a second-year university class at Utrecht University on functional programming in September 2013. Each of these log entries consists of:

—an IP address

—a user name

—a requested service: a hint, a list of exercises, or the diagnosis of a submitted student program

We are particularly interested in the diagnosis requests. 3466 log entries request to diagnose a student program. We will call these log entries interactions. Besides the above components and some more administrative information, such as the version of Ask-Elle used, these interactions consist of:

—a name of a programming exercise (such as $dupli$ or $repli$)

—a student program

—the result of the diagnosis of the student program. The diagnose service reports that there is a syntax or a type error, that the student program can be completed into a model solution, that the student has finished the exercise, that there is a counterexample for the student program, or that it cannot diagnose the student program.

[5]http://community.haskell.org/~ndm/hlint/

Category	Attempts		Interactions	
Compiler error	142	(21.8%)	1920	(55.4%)
Model	221	(33.9%)	754	(21.8%)
Counter	33	(5.1%)	201	(5.8%)
Tests passed	235	(36.0%)	436	(12.6%)
Discarded	21	(3.2%)	155	(4.5%)
Total	652		3466	

Table I. Categorising student programs

Category	Attempts		Interactions	
Compiler error	44	(31.2%)	508	(63.8%)
Model	65	(46.1%)	184	(23.1%)
Counter	4	(2.8%)	27	(3.4%)
Tests passed	27	(19.1%)	68	(8.5%)
Discarded	1	(0.7%)	9	(1.1%)
Total	141		796	

Table II. Categorising interactions for $dupli$

Category	Attempts		Interactions	
Compiler error	12	(18.5%)	275	(67.2%)
Model	12	(18.5%)	40	(9.8%)
Counter	6	(9.2%)	15	(3.7%)
Tests passed	31	(47.7%)	62	(15.1%)
Discarded	4	(6.2%)	17	(4.2%)
Total	65		409	

Table III. Categorising interactions for $repli$

The 3466 interactions with Ask-Elle come from 116 out of the 285 students registered for the course. Students seemed to work top-down through the list of exercises: the exercises $dupli$, $range$, and $repli$ have been tried quite a lot; exercises that appear at the bottom of the exercise list have been tried much less. In total, the students worked on 26 different programming exercises. The log entries have been grouped into *exercise attempts*: sequences of interactions resulting in either a solution to the exercise, or the student giving up on the exercise. On average, students worked on 5.62 exercise attempts (standard deviation 6.57). An exercise attempt consists on average of 5.29 interactions (standard deviation 6.12). We have divided the entire set of interactions and exercise attempts in the categories given in the previous section. To classify an attempt we use its last interaction. The overall results are shown in Table I. The results for the functions for which we received the most interactions, $dupli$, $repli$, and $compress$, are shown in Tables II, III, and IV.

We want to recognise as many correct programs as possible with the model solutions. We define the ratio of recognised model solutions by Ask-Elle:

$$recognised = \frac{|Model|}{|Model| + |Tests\ passed| + |Discarded|}$$

Note that this ratio is a lower bound: there may be undetected incorrect solutions in the *Tests passed* and *Discarded* classes. Programs with a compiler error or for which a counterexample is found are incorrect and thus excluded in this ratio. Currently, 56.1% of the interactions (and 46.3% of the attempts) are recognised to be correct.

Category	Attempts		Interactions	
Compiler error	19	(31.2%)	270	(56.4%)
Model	11	(18.0%)	104	(21.7%)
Counter	4	(6.6%)	26	(5.4%)
Tests passed	24	(39.3%)	47	(9.8%)
Discarded	3	(4.9%)	32	(6.7%)
Total	61		479	

Table IV. Categorising interactions for *compress*

Similarly, we define the ratio of *classified* correct or incorrect programs by:

$$\mathit{classified} = \frac{|\mathit{Model}| + |\mathit{Error}| + |\mathit{Counter}|}{|\mathit{Total}|}$$

Of all interactions, 82.9% is classified as correct or incorrect. Of all attempts, 60.7% is classified as correct or incorrect.

Some observations about the data:

—The number of syntax and type errors is high, even in completed exercise attempts. In 21.8% of the exercise attempts, a student gave up on the exercise with a compiler error in her last submission.

—Ask-Elle scores better on individual interactions. However, many of the recognised inputs are relatively small and largely incomplete: input such as $rotate\ ?\ ? = ?$ is classified as *Model*.

—Possible reasons for why the results for $dupli$ are better than the results for $repli$ are that there was a bug in alpha-conversion (found during and repaired after the experiment) which would fire sooner in definitions with two variables instead of one, and that the number of specified model solutions for $dupli$ is higher than for $repli$ (6 versus 4).

To increase the *recognised* and *classified* ratios, we analyse the set of programs in *Tests passed* and *Discarded* to discover which program transformations or model solutions we should add. The *recognised* ratio is also increased by improving or adding properties that are used to find counterexamples for erroneous solutions, because then the number of elements in the *Tests passed* and *Discarded* classes decrease.

5.1 Program transformations

We show which program transformations would move student programs from the *Tests passed* category to the *Model* category by example. We illustrate the transformations using the $encode$ exercise, which asks a student to define a function that returns the 'run-length encoding' of a list:

$$encode :: Eq\ a \Rightarrow [a] \rightarrow [(Int, a)]$$

For example:

$$> encode\ [1,2,2,3,2,4]$$
$$[(1,1),(2,2),(1,3),(1,2),(1,4)]$$

One of the model solutions for $encode$ uses the prelude function $takeWhile$ (or $dropWhile$), which takes (drops) elements from the front of a list as long as a given property holds.

$$\begin{aligned} encode_m\ [\,] &= [\,] \\ encode_m\ (x : xs) &= (n+1, x) : encode_m\ (drop\ n\ xs) \\ &\quad \textbf{where}\ n = length\ (takeWhile\ (==x)\ xs) \end{aligned}$$

We use the subscripts m and s to distinguish the model solution from a student program. The following student program is very similar to the model solution:

$$\begin{aligned} encode_s\ [\,] &= [\,] \\ encode_s\ (x : xs) &= (length\ \$\ x : takeWhile\ (==x)\ xs, x) \\ &\quad :\ encode_s\ (dropWhile\ (==x)\ xs) \end{aligned}$$

The student program does not use a where-clause. Values defined in a where-clause should be inlined. From the model solution we obtain the following program by replacing the constant n with its definition:

$$\begin{aligned} encode_m\ [\,] &= [\,] \\ encode_m\ (x : xs) &= (length\ (takeWhile\ (==x)\ xs) + 1, x) \\ &\quad : encode_m\ (drop\ (length\ (takeWhile\ (==x)\ xs))\ xs) \end{aligned}$$

The student program uses the $(\$)$ operator, Haskell's explicit (right-associative) application operator, to avoid writing parentheses. One of Ask-Elle's normalisations removes occurrences of this operator. Furthermore, since $length$ is a standard recursive function on lists, the student program can be beta-reduced. Using $(\$)$-removal and beta-reduction, we get:

$$\begin{aligned} encode_s\ [\,] &= [\,] \\ encode_s\ (x : xs) &= (1 + length\ (takeWhile\ (==x)\ xs), x) \\ &\quad :\ encode_s\ (dropWhile\ (==x)\ xs) \end{aligned}$$

Two other normalisation steps Ask-Elle employs are removing redundant parentheses and ordering the arguments of commutative operators such as $(+)$ and $(\wedge)$ using some (arbitrary but consistent) choice based on the lexicographical order of the arguments. Applying these steps results in:

$$\begin{aligned} encode_m\ [\,] &= [\,] \\ encode_m\ (x : xs) &= (1 + length\ (takeWhile\ (==x)\ xs), x) \\ &\quad : encode_m\ (drop\ (length\ (takeWhile\ (==x)\ xs))\ xs) \end{aligned}$$

To transform this program into one that is the same as the (transformed) student program, we use the fact that $drop\ (length\ (takeWhile\ (==x)\ xs))\ xs$ and $dropWhile\ (==x)\ xs$ are *semantically* equivalent. Such equivalences cannot be generated automatically, but they can be specified as rewrite rules together with the model solutions [Gerdes et al. 2012]:

$$\begin{aligned} &\{\text{-\# ALT} \\ &\quad dropWhile\ p\ xs = drop\ (length\ (takeWhile\ p\ xs))\ xs\ \text{\#-}\} \end{aligned}$$

Using this rewrite rule, we can infer that the student program is equivalent to the model solution.

5.2 Program transformation for student programs

For all student programs in the *Tests passed* category, we determine whether or not they can be recognised if we would add to or improve program transformations in Ask-Elle. We collect a list of program transformations that help to recognise student programs. Besides program transformations, we have also investigated which programs require a new model solution, which programs contain imperfections, and whether or not programs have the correct input-output behaviour. With this information we answer the questions of Section 4.

Below we list the program transformations that have to be added or improved to match a student program to a model solution. Note that a student program might require multiple transformations, require a new model solution, and contain multiple imperfections.

(1) Many students include type signatures in their programs. Although this is of course good practice, our tutor does not recognise type signatures when matching against a model solution. This is problematic for 94 student programs, many of which are also unrecognised for other reasons.
(2) Recognising more functions from the prelude and adding alternative definitions for prelude functions helps in 37 cases. Using function definitions to perform a beta-reduction step helps in 39 cases.
(3) Dealing with function parameters uniformly. A student program such as $palindrome = (\lambda x \to x == reverse\ x)$ for the *palindrome* exercise is not matched against the model solution $palindrome\ x = x == reverse\ x$. Or a student does not use function composition, as in $dupli\ x = concatMap\ (replicate\ 2)\ x$. Often some variant of eta-conversion, for example replacing $(\lambda x \to (+)\ 1\ x)$ by $(+)\ 1$ is sufficient. The latter kind of examples typically appear in the function argument of a higher-function. There are 8 + 54 + 13 = 75 occurrences of these transformations. We expect such programs can be recognised by introducing eta-conversion and normalising all programs to lambda-expressions.
(4) The alpha-renaming normalisation step contained a bug. This problem appears in 48 programs. An additional 19 programs are not recognised due to the use of a wildcard pattern in either the student program or in the model solution (similar to the student program).
(5) Inlining a value defined in a **where**-clause, a **let**-clause, or a separate top-level definition helps in 26 cases.
(6) If an expression is guarded by an equality such as $a == b$, we can replace all occurrences of a by b (or b by a) in the expression. In 26 cases this transformation helps.
(7) In 22 cases a program desugaring, such as converting the Haskell list-notation to constructor application, would help.
(8) One program requires removing an unused (helper) definition. We cannot remove an unused helper function in an incomplete program, because such a definition may be called when a hole is further refined. The same problem holds for inlining helper-definitions.
(9) Many more types of required transformations appear very infrequently, such as: removing infix-notation, changing the order of arguments of a helper function, changing the order of patterns, transforming between guards, patterns and if-then-else constructions, changing the conditions in guards (for example changing $x \not\equiv y$ into $x == y$, including changing the code after the guard) and more.

Besides these transformations, we also found that it is sometimes worthwhile to introduce a more abstract version of a model solution to increase the number of student programs that are recognised. For example, a number of our exercises require recursing over integers until a stop condition is met. Consider the *range* exercise, in which a student should define a function that enumerates all numbers in a given range. For instance, $range\ 2\ 5$ gives $[2,3,4,5]$. We may assume the second integer to be larger than the first. Here are some correct and equivalent definitions:

$$\begin{array}{lll} range_1\ a\ b & | & a == b \quad = [a] \\ & | & otherwise = a : range_1\ (a+1)\ b \end{array}$$

$$\begin{array}{lll} range_2\ a\ b & | & a == b \quad = [b] \\ & | & otherwise = a : range_2\ (a+1)\ b \end{array}$$

$$\begin{array}{lll} range_3\ a\ b & | & a > b \quad = [\,] \\ & | & otherwise = a : range_3\ (a+1)\ b \end{array}$$

$$\begin{array}{lll} range_4\ a\ b & | & a > b \quad = [\,] \\ & | & otherwise = range_4\ a\ (b-1) +\!\!+ [b] \end{array}$$

$$\begin{array}{lll} range_5\ a\ b & | & a \leqslant b \quad = a : range_5\ (a+1)\ b \\ & | & otherwise = [\,] \end{array}$$

The first definition can be transformed in the second definition by means of program transformation 6. The other definitions show various ways in which the arguments a and b can be used to steer the recursion: going up from a to b, or down from b to a. The use of $==$, $>$, or $\leqslant$ in guards increases the number of variants, and there are many other constructs that introduce variants. Just as *foldr* can be used to recognise uses of both *foldr* itself as well as its explicitly recursive variants, we expect that many of the variants of the *range* function can be inferred from a sufficiently abstract definition for this exercise, such as

$$\begin{array}{l} cond\_iterate\ cond\ begin\ it\ af\ bf\ a\ b \\ \quad | \ cond\ a\ b \ = begin \\ \quad | \ otherwise = it\ a\ b\ step \\ \quad \textbf{where} \\ \qquad step = cond\_iterate\ cond\ begin\ it\ af\ bf\ (af\ a)\ (bf\ b) \end{array}$$

$$\begin{array}{l} range\ a\ b = \\ \quad cond\_iterate\ (>)\ [\,]\ (\lambda a'\ \_\ acc \to a' : acc)\ (+1)\ id\ a\ b \end{array}$$

We have yet to investigate the kind of program transformations necessary to use this approach.

5.3 Results

We return to the questions posed in Section 4.

—How many programs are classified as undecided? 17.1% of all interactions and 39.2% of all attempts end in *Undecided*. These results are better for smaller assignments with many model solutions, such as *dupli*.
—How often would improving or adding a program transformation help with matching against model solutions? By adding new transformations Ask-Elle recognises 161 programs that pass the tests as model solutions. By fixing the alpha-renaming transformation and improving other transformations Ask-Elle can recognise an additional 96 programs.
—How often would adding a model solution help? Of the remaining $436-161-96 = 179$ programs in the *Tests passed* category, we expect to recognise 84 programs by adding more model solutions. Note that to recognise some of these programs, we need the improved or new program transformations from the previous point. For 16 of the 26 exercises on which students worked we need one or more new model solutions. Three of these were used in ten or more student programs.
—How often do students add irrelevant, with respect to the exercise, parts to a program with the correct input-output behaviour? In 3 programs a student deals with cases that are excluded in the definition of the exercise, for example a case for negative numbers in an exercise that states that the input number is at least zero.
—How many of the programs with correct input-output behaviour contain imperfections, such as redundant case-clauses, or an inefficient implementation? We have found 86 such programs, including the 3 from the previous point. These programs contain superfluous patterns or cases (20), for example for the empty list, the singleton list, and a cons pattern, where the singleton

Category	Interactions	
Compiler error	1920	(55.4%)
Model	1095	(31.6%)
Counter	206	(5.9%)
Tests passed	87	(2.5%)
Discarded	158	(4.6%)
Total	3466	

Table V. Categorising student programs

pattern is covered by the cons pattern and the empty pattern, or a helper function that directly (18) or indirectly (4) corresponds to a prelude function, such as an instance of map without a function argument that applies a particular function to each value in a list. Some students delay pattern-matching (25), for example by using $head$ and $tail$ instead of using the (:) constructor for lists.

—How often does QuickCheck not find a counterexample, although the student program is incorrect? The remaining 179 - 84 - 86 = 9 programs are incorrect, but QuickCheck does not find a counterexample. For the incorrect programs that contain holes (3) there is no way to fill the holes to obtain a correct program, but QuickCheck will not find counter-examples to these programs. Since most of the tests are discarded these programs end up in the *Discarded* category. An example of such a program is $dupli\ xs = map\ ?\ xs$. This definition is correct for the input $[\,]$, but will be incorrect for any nonempty input. However, all tests with nonempty inputs are discarded. This error can maybe be caught by deriving properties from student programs, and determining that model solutions do not satisfy these properties. This is probably not easily added to Ask-Elle. The incorrect programs without holes (6) require more precise properties for the corresponding exercises. For the program that is a non-solution to the $primes$ exercise (1) this is hard. The $primes$ exercise asks for an infinite list of prime numbers, and testing the student program, which also returns an infinite list, against the model solution and determining an error might take a long time. It takes an infinite amount of time if the student program happens to be correct. We decided to only check the first 100 elements of the list of primes returned by the student. The first 100 elements of the list returned by the erroneous student program happened to be correct.

We give the version of Table I for interactions, taking the new and improved program transformations and new model solutions into account, in Table V. We move 161 + 96 + 84 = 341 from *Test passed* to *Model* and move the 3 incorrect programs with holes to *Discarded* and the 5 incorrect programs without holes for which we can update the properties of the exercise to *Counter*.

Thus the *recognised* ratio of interactions increases to 81.7% (was: 56.1%), and the *classified* ratio to 92.9% (was: 82.9%).

6. FUTURE WORK

Ask-Elle diagnoses a student program to be correct (transformable to a model solution), or incorrect (together with a counterexample). A teacher sometimes also gives more subtle feedback such as: this is good solution, but it is better to ... We want to draw up a feedback benchmark, in which we collect the kind of feedback that is usually given by teachers on the kind of functional programs that are offered in Ask-Elle, and we want to study if we can incorporate this kind of feedback in Ask-Elle, for example by specifying undesirable transformations we may perform on a student program to transform it to a model solution, and reporting these.

Finally, we want to determine and implement more program transformations for recognising student programs, and use our improved tutor in a new experiment.

7. CONCLUSIONS

We have studied the feasibility of using matching against model solutions and testing against program properties for diagnosing possibly incomplete student programs in a tutoring system for the programming language Haskell. We classified almost 3500 student programs logged by our tutor, and found that we could diagnose 82.9% of the (correct or incorrect) student submissions. The feedback from Ask-Elle is sound, but incomplete, that is, a teacher can sometimes give feedback that is not given by Ask-Elle. By analysing the student programs, we collected a list of missing program transformation, model solutions, and properties of model solutions that would increase the number of diagnosed programs to 92.9%. Besides a set of new program transformations, properties of model solutions, and model solutions, the experiment also led to a deeper understanding of the relation between model solutions, program transformations, and teacher annotations, and improved the quality of the exercises offered in Ask-Elle.

ACKNOWLEDGMENTS

We thank Jurriaan Hage for allowing us to perform an experiment with Ask-Elle in his classes, and Tom Tervoort and Gabe Dijkstra for contributing to the source code of Ask-Elle. Anonymous reviewers on previous versions of this paper provided helpful comments.

REFERENCES

Kirsti M Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15, 2 (2005), 83–102.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of ICFP 2000: International Conference on Functional Programming*. ACM, 268–279.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2010. Using strategies for assessment of programming exercises. In *Proceedings of SIGCSE 2010: the 41st ACM technical symposium on Computer science education*, Gary Lewandowski, Steven A. Wolfman, Thomas J. Cortina, and Ellen Lowenfeld Walker (Eds.). ACM, 441–445.

Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2012. An Interactive Functional Programming Tutor. In *Proceedings of ITICSE 2012: the 17th Annual Conference on Innovation and Technology in Computer Science Education*, T. Lapidot, J. Gal-Ezer, M.E. Caspersen, and O. Hazzan (Eds.). ACM, 250–255.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of Haskell 2003: the ACM SIGPLAN Workshop on Haskell*. ACM, 62–71.

Johan Jeuring, Alex Gerdes, and Bastiaan Heeren. 2011. A programming tutor for Haskell. In *Proceedings of CEFP 2011: Lecture Notes of the Central European School on Functional Programming*, Viktória Zsók, Zoltan Horváth, and Rinus Plasmeijer (Eds.). LNCS, Vol. 7241. Springer, 1–45.

Wei Jin, Tiffany Barnes, John C. Stamper, Michael John Eagle, Matt Johnson, and Lorrie Lehmann. 2012. Program Representation for Automatic Hint Generation for a Data-Driven Novice

Programming Tutor. In *Proceedings of ITS 2012: the 11th International Conference on Intelligent Tutoring Systems*, Stefano A. Cerri, William J. Clancey, Giorgos Papadourakis, and Kitty Panourgia (Eds.). LNCS, Vol. 7315. Springer, 304–309.

Simon Peyton Jones et al. 2003. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press.

Kelly Rivers and Kenneth R. Koedinger. 2014. Automating Hint Generation with Solution Space Path Construction. In *Proceedings of ITS 2014: the 12th International Conference on Intelligent Tutoring Systems*, Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha Crosby, and Kitty Panourgia (Eds.). LNCS, Vol. 8474. Springer, 329–339.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of PLDI 2013: the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 15–26.

Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist* 46, 4 (2011), 197–221.

Jeroen Voeten. 2001. On the Fundamental Limitations of Transformational Design. *ACM Transactions on Design Automation of Electronic Systems* 6, 4 (2001), 533–552.

Songwen Xu and Yam San Chee. 2003. Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. *IEEE Transactions on Software Engineering* 29, 4 (2003), 360–384.