Noname manuscript No.
(will be inserted by the editor)

Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback

Alex Gerdes · Bastiaan Heeren · Johan Jeuring · L. Thomas van Binsbergen

Received: date / Accepted: date

Abstract Ask-Elle is a tutor for learning the higher-order, strongly-typed functional programming language Haskell. It supports the stepwise development of Haskell programs by verifying the correctness of incomplete programs, and by providing hints. Programming exercises are added to Ask-Elle by providing a task description for the exercise, one or more model solutions, and properties that a solution should satisfy. The properties and model solutions can be annotated with feedback messages, and the amount of flexibility that is allowed in student solutions can be adjusted.

The main contribution of our work is the design of a tutor that combines (1) the incremental development of different solutions in various forms to a programming exercise with (2) automated feedback and (3) teacher-specified programming exercises, solutions, and properties. The main functionality is obtained by means of strategy-based model tracing and property-based testing. We have tested the feasibility of our approach in several experiments, in which we analyse both intermediate and final student solutions to programming exercises, amongst others.

Keywords Functional programming · Haskell · Tutoring · Model tracing · Automated feedback · Adaptability

Alex Gerdes
QuviQ and Chalmers University of Technology
E-mail: agerdes@me.com

Bastiaan Heeren
Open Universiteit Nederland
E-mail: Bastiaan.Heeren@ou.nl

Johan Jeuring
Utrecht University and Open Universiteit Nederland
E-mail: J.T.Jeuring@uu.nl

L. Thomas van Binsbergen
Royal Holloway, University of London
E-mail: ltvanbinsbergen@acm.org

1 Introduction

Haskell is a lazy, purely functional programming language (Peyton Jones, 2003). It is taught at universities all over the world: just the English Haskell beginners' books (Bird, 1998; Hutton, 2007; Hudak, 2000; Thompson, 2011) together sold more than 50, 000 copies, and there are lecture notes and books available in Spanish, Dutch, German, Portuguese, Russian, Japanese, etc.

Ask-Elle[1] is a tutor that supports the stepwise development of simple functional programs in Haskell. 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 can have a look at a worked-out solution. Ask-Elle is an example of an intelligent tutoring system (VanLehn, 2006) for the domain of functional programming.

Why would a teacher or a student use such an intelligent tutoring system? Evaluation studies have indicated that

- using an intelligent tutor that supports the stepwise development of solutions to problems is almost as effective as a human tutor (VanLehn, 2011),
- working with an intelligent tutor supporting the construction of programs is more effective when learning how to program than doing the same exercise "on your own" using only a compiler, or just pen-and-paper (Corbett et al., 1988),
- using intelligent tutors requires less help from a teacher while showing the same performance on tests (Odekirk-Hash and Zachary, 2001),
- using such tutors increases the self-confidence of female students (Kumar, 2008), and
- the immediate feedback given by many of the tutors is to be preferred over the delayed feedback common in classroom settings (Mory, 2003).

The type of exercises that a learning environment supports determines to a large extent how difficult it is to generate feedback. For example, it is much easier to specify feedback for a multiple-choice exercise, than for an essay. Le and Pinkwart (2014) propose a classification of programming exercises supported in learning environments. They base their classification on the degree of ill-definedness of a programming problem. Class 1 exercises have a single correct solution. Examples are quiz-like questions with a single solution, or slots in a program that need to be filled in to complete some task. Class 2 exercises can be solved by different implementation variants. Usually a program skeleton or other information that suggests the solution strategy is provided, but variations in the implementation are allowed. Finally, class 3 exercises can be solved by applying alternative solution strategies. Independently developing a program solving a class 3 exercise, is an important learning objective for learning programming (Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) and IEEE Computer Society, 2013). Ask-Elle offers class 3 exercises.

Teachers need control over the learning environment they use: adaptability of learning environments is important (Anderson et al., 1995; Bokhove and Drijvers, 2010). Ask-Elle offers a teacher a lot of flexibility in adding new programming exercises to the tutor, and in adapting or specialising the feedback. The feedback and hints provided by Ask-Elle are calculated automatically from teacher-specified, annotated solutions and properties for a programming exercise.

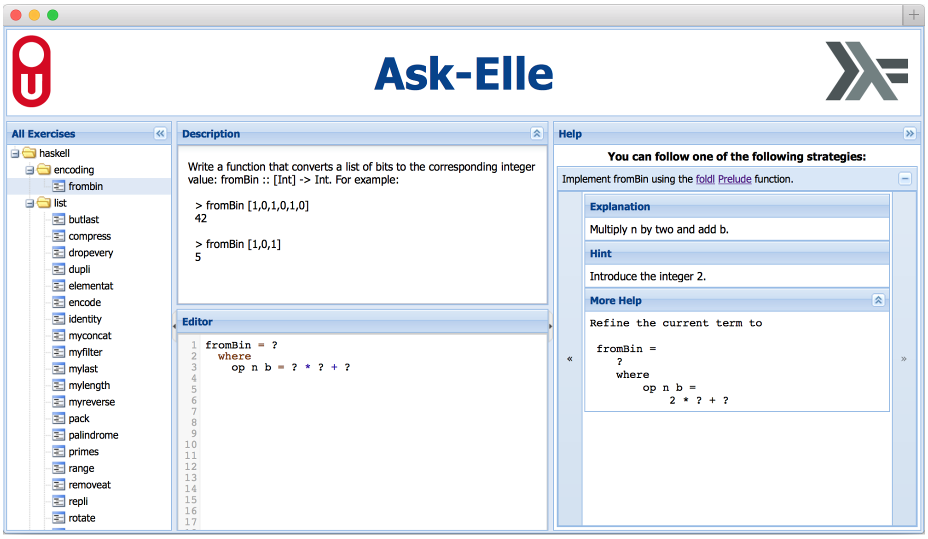
[1] `http://ideas.cs.uu.nl/FPTutor/`

Programming tutors have been built since the 1970s, for programming languages such as Lisp (Anderson et al., 1986), Prolog (Hong, 2004), Java (Holland et al., 2009), Pascal (Johnson and Soloway, 1985), C (Wang et al., 2011), and many more. None of these tutors supports automatic feedback on the incremental development of programs for class 3 exercises, in which teachers can easily add programming exercises and adapt feedback. For example, the Lisp tutor (Anderson et al., 1986) does give feedback on the steps a student takes towards a solution to a programming exercise, and allows students to solve a program flexibly in that students do not have to follow a strict top to bottom, left to right order (Corbett et al., 1988), but it offers class 2 exercises. Also, adding an exercise to the Lisp tutor is non-trivial. Successors of the Lisp tutor, such as ELM-ART (Brusilovsky et al., 1996), still offer class 2 exercises. As another example, the Prolog tutor (Hong, 2004) does offer class 3 programming exercises, but forces a student to first select the kind of solution she wants to write, and only then allows the development of a program. As a final example, AutoLEP (Wang et al., 2011), which is an program assessment system rather than a programming tutor, offers class 3 exercises, but cannot deal with intermediate, incomplete programs.

The main contribution of this paper is the design of a programming tutor that offers class 3 programming exercises, supports the incremental development of solutions to such exercises, and automatically calculates feedback and hints. These feedback and hints are derived automatically from teacher-specified annotated solutions for a problem. The main functionality of the tutor is obtained by means of strategy-based model tracing (Heeren et al., 2010), and property-based testing (Claessen and Hughes, 2000). Furthermore, we use quite a bit of compiler technology for functional programming languages to offer as much flexibility as possible. We test the feasibility of our approach in several experiments, in which we analyse both intermediate and final student solutions to programming exercises to find out how well our tutor deals with possibly incomplete student programs, amongst others.

This paper is organised as follows. Section 2 introduces Ask-Elle by means of an example session of a student interacting with Ask-Elle, and shows what a teacher has to do to add a programming exercise to the tutor. This section exemplifies our research contributions. Section 3 shows the design of Ask-Elle, and contains our main research contribution: it gives the architecture of Ask-Elle, and briefly discusses the technologies we use for diagnosing student programs. After the sections that introduce Ask-Elle, we describe three experiments we have performed with Ask-Elle. Section 4 shows an experiment in which we use Ask-Elle for assessing a lab exercise, and Section 5 describes a class experiment in which we use a questionnaire to find out how students experience the use of Ask-Elle. After performing these experiments we decided on a classification of student programs, which we describe in Section 6. Using this classification, Section 7 describes an experiment in which we analyse the quality of the feedback given by Ask-Elle. We discuss related work in Section 8. Section 9 discusses future work and concludes the paper.

This paper combines and revises results described in several earlier workshop and conference papers and demonstrations (Gerdes et al., 2010, 2012a,b; Jeuring et al., 2012, 2014).



Write a function that converts a list of bits to the corresponding integer value: fromBin :: [Int] -> Int. For example:

```
> fromBin [1,0,1,0,1,0]
42

> fromBin [1,0,1]
5
```

```
fromBin = ?
  where
    op n b = ? * ? + ?
```

You can follow one of the following strategies:

Implement fromBin using the foldl Prelude function.

Explanation

Multiply n by two and add b.

Hint

Introduce the integer 2.

More Help

```
Refine the current term to

 fromBin =
    ?
    where
        op n b =
            2 * ? + ?
```

Figure 1 The web-based functional programming tutor

2 An example session

This section introduces Ask-Elle by means of an example session, in which a student develops a program in Ask-Elle and uses several of its features. Then it shows how a teacher can add an exercise to Ask-Elle and adapt the feedback and behaviour of Ask-Elle.

2.1 An example student session in Ask-Elle

We demonstrate Ask-Elle by showing some interactions of a hypothetical student with the functional programming tutor. A screenshot of Ask-Elle is shown in Figure 1. Ask-Elle sets small functional programming tasks, and gives feedback in interactions with the student. We assume that the student has attended lectures on how to write simple functional programs on lists.

At the start of a tutoring session the tutor gives a problem description. Here the student has to write a program for converting a list of binary numbers to its decimal representation.

`Write a function that converts a list of bits to the corresponding decimal value:`

$$\mathit{fromBin} :: [\mathit{Int}] \rightarrow \mathit{Int}$$

`For example:`

$$> \mathit{fromBin}\ [1,0,1,0,1,0]$$
$$42$$
$$> \mathit{fromBin}\ [1,0,1]$$
$$5$$

The tutor displays the following starting point:

$\bullet$

The task of a student is to refine the holes, denoted by $\bullet$, of the program. The starting point of an exercise is a single hole. After each refinement, a student can ask the tutor whether or not the refinement is bringing her closer to a correct solution. If a student does not know how to proceed, she can ask the tutor for a hint. A student can also introduce new declarations, function bindings, and alternatives.

Suppose the student has no idea where to start and asks the tutor for help. The tutor offers several ways to help. For example, it can list all possible ways to proceed solving an exercise. In this case, the tutor responds with:

```
You can proceed in several ways:
- Implement fromBin using the foldl prelude[2] function.
- Take the inner product with a list of factors of two.
- Implement fromBin with a helper function using an extra
parameter.
```

We assume that a student has already been exposed to the functions and concepts mentioned in the feedback, for example via lectures, a teaching assistant, or lecture notes. However, the prelude functions in the feedback are linked to external web pages with detailed information, such as their type signature and example usages.

The tutor can make a choice between the different possibilities, so if the student does not want to choose, and just wants a single hint, she gets:

```
Implement fromBin using the foldl prelude function.
```

Here we assume that the teacher has set up the tutor to prefer the solution that uses *foldl*, defined by:

$$\begin{aligned}
&\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a \\
&\mathit{foldl}\ \mathit{op}\ e\ [\,] \qquad\ = e \\
&\mathit{foldl}\ \mathit{op}\ e\ (x : \mathit{xs}) = \mathit{foldl}\ \mathit{op}\ (\mathit{op}\ e\ x)\ \mathit{xs}
\end{aligned}$$

Let us briefly explain this function before we continue with the session. The higher-order function *foldl* processes a list using an operator that associates to the left (Hutton, 2007). Its definition covers two cases: one for the empty list $[\,]$, and the case of one element (x) in front of the rest of the list (xs). Consider the list of bits from the session's example: $[1,0,1,0,1,0]$. Given some binary operator $\oplus$ and a start value e, *foldl* turns this list into the expression:

$$((((((e \oplus 1) \oplus 0) \oplus 1) \oplus 0) \oplus 1) \oplus 0$$

Now if we let $n \oplus b = 2 * n + b$ and $e = 0$, we can indeed calculate the expected result (42) from the above expression.

The student starts by introducing the function name as asked by the exercise, but postpones the definition of the function body by introducing another hole:

$$\mathit{fromBin} = \bullet$$

The student can ask for more detailed information at this point about the body of the function, and the tutor responds with giving increasing detail:

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

`Define the` *fromBin* `function using` *foldl*`. The operator should multiply the intermediate result with two and add the value of the bit.`

with the final bottom-out hint:

`Define:`

$$fromBin = foldl \bullet \bullet$$

At this point, the student can refine the function at two positions. We do not impose an order on the sequence of refinements. Moreover, a student can refine multiple holes at once:

```
fromBin = foldl op •
  where
    op • • = •
```

In the above step the student has made two refinements: she refined the operator hole to the variable *op*, and introduced a function binding for that variable in a **where** clause. The student continues with the operator and introduces two pattern variables:

```
fromBin = foldl op •
  where
    op n b = •
```

which is accepted by the tutor. She continues with the definition of the operator:

```
fromBin = foldl op •
  where
    op n b = • + •
```

If the student now asks for a hint, the tutor responds with:

`Multiply` *n* `by two and then add` *b*`.`

She continues with:

```
fromBin = foldl op •
  where
    op n b = 2 * n + c
```

which gives:

`Error: undefined variable` *c*

This is a syntax-error message generated by the Helium compiler (Heeren et al., 2003), which we use in our tutor. Helium gives excellent syntax-error and type-error messages, and reports dependency analysis problems in a clear way. The student corrects the syntax-error and continues with:

```
fromBin = foldl op 1
  where
    op n b = 2 * n + b
```

which results in the following feedback message from the tutor:

```
Your implementation is incorrect for the following input: []
We expected 0, but we got 1
```

The definition does not match any of the model solutions, and by means of random testing the tutor can find an example where the result of the student program invalidates a teacher-specified property or differs from a model solution. We use this example to generate the detailed feedback message. Correcting the error, the student enters:

$$
\begin{array}{l}
\mathit{fromBin} = \mathit{foldl}\ \mathit{op}\ 0 \\
\quad \textbf{where} \\
\qquad \mathit{op}\ n\ b = 2 * n + b
\end{array}
$$

which completes the exercise.

These interactions show that our tutor can give hints about which step to take next, in various levels of detail, list all possible ways in which to proceed, point out errors and pinpoint where the error appears to be, and show a complete worked-out example. This subsection exemplifies our main research contribution: a programming tutor that offers class 3 programming exercises, supports the incremental development of solutions to such exercises, and automatically calculates feedback and hints.

2.2 Teacher

This subsection demonstrates how a teacher adds a programming exercise to the tutor, and adapts the feedback given by the tutor. An exercise is specified by a set of *model solutions* to the programming problem and a configuration file, grouped together in a directory. A configuration file is a Haskell source file that contains a task description together with a list of QuickCheck (Claessen and Hughes, 2000) properties for the exercise. Each model solution is specified in a separate file. The interactions of the tutor are based on these model solutions and properties.

A teacher can group exercises together, for example for practising list problems, collecting exercises of the same difficulty, or exercises from a particular textbook.

2.2.1 Model solutions

A model solution is a program that an expert writes, using good programming practices. We have specified three model solutions for *fromBin* (see Figure 2). We already explained the first model solution that is based on *foldl*.

The second model solution reverses the input list, and then computes the inner product of this list and a list of powers of two. The definition consist of three parts, which are combined using function composition ($\circ$). Function composition takes two functions as argument and applies the left-hand side function to the output of the right-hand side function. The first part of this solution reverses the input list, which turns the list in least significant bit first order. The second part creates an infinite list of powers of two by using the *iterate* function:

$$\mathit{iterate}\ (*2)\ 1 = [1, 2, 4, 8, 16, \ldots]$$

```
  -- 1. Solution with foldl
fromBin = foldl op 0
    where
      op n b = 2 ∗ n + b
  -- 2. Inner product with powers of two
fromBin = sum ◦ zipWith (∗) (iterate (∗2) 1) ◦ reverse
  -- 3. Tupling, passing around the length as extra argument
fromBin bs = fromBin′ (length bs − 1) bs
  where
    fromBin′ _ []       = 0
    fromBin′ n (b : bs) = b ∗ 2 ^ n + fromBin′ (n − 1) bs
```

Figure 2 Three model solutions for the *fromBin* programming exercise

The powers of two are multiplied element-wise with the reversed list using *zipWith*. The third and final step is to add all elements using the *sum* function. Note that this solution relies on lazy evaluation for dealing with the infinite list. The *zipWith* function discards excess elements if one list is longer than the other.

The final model solution uses a helper function *fromBin*′, which takes as extra parameter the length of the input list (n) for calculating the correct power of two. The value of n is decremented by one at every recursive call. The advantage of this extra parameter is that we only have to calculate the length of the list once, instead of at each recursive call. Passing around extra, intermediate results is a well-known programming technique called tupling.

The tutor uses these model solutions to generate feedback. It recognises many variants of a model solution. For example, the following solution:

```
fromBin xs = let f z []       = z
                 f z (x : xs) = f (base ∗ z + x) xs
                 base         = 2
                 start        = 0
             in f start xs
```

is recognised from the first model solution. To achieve this, we not only recognise the usage of a prelude function, such as *foldl*, but also its definition. Furthermore, we apply a number of program transformations to transform a program to a normal form.

2.2.2 Adapting feedback

A teacher adapts the feedback given to a student by *annotating* model solutions. An annotation is done via a pragma, which is a special kind of source code comment. Using the following construction a teacher adds a *description* to a model solution:

```
{-# DESC Implement fromBin using the foldl prelude function. #-}
```

The first hint in Section 2.1 gives the descriptions for the three model solutions for the *fromBin* exercise. The appearances of prelude functions in a hint are hyperlinked to web pages with detailed information. These links are generated automatically. More fine-grained and location specific feedback can be added to a model solution using the `FB` feedback annotation:

```
fromBin =
  {-# FB Define the fromBin function using foldl. The op... #-}
  foldl op 0
    where
      op n b = {-# FB Multiply n by two and add b. #-}2 * n + b
```

Thus we can give a detailed explanation of how to construct the operator *op*. These feedback messages are organised in a hierarchy based on the abstract syntax tree of the model solution. This enables the teacher to configure the tutor to give feedback messages with an increasing level of detail.

The `DESC` and `FB` annotations can be used to steer the textual feedback given by Ask-Elle to a student. In addition to these textual feedback annotations, Ask-Elle offers two annotations that a teacher can use to control the variation of accepted solutions. We mentioned before that we not only recognise the usage of a library function, such as *foldl*, but also the definition of that function. When a student is not aware of a particular library function and defines such a function herself, we can recognise and approve it. Sometimes, however, a teacher wants a student to use a library function instead of her own definition. To enforce the usage of a library function, in this case *iterate*, a teacher can annotate the function with the `MUSTUSE` annotation:

```
fromBin = sum ∘ zipWith (*) ({-# MUSTUSE #-}iterate (*2) 1) ∘ reverse
```

The `MUSTUSE` annotation limits the variation of allowed implementations. A teacher can also introduce an alternative implementation for a library function to allow more implementations by means of the `ALT` annotation, of which we give an example in Section 4.1.

2.2.3 Configuration file

A teacher needs to create a *configuration file* (besides model solutions) to define an exercise. This configuration file is a Haskell source file that contains a global description of the exercise and a list of QuickCheck properties. We reuse the `DESC` annotation for the global description that introduces the programming problem. For example:

```
{-# DESC Write a function that converts a list of bits to the
corresponding decimal value: fromBin::[Int] → Int. For example:

   > fromBin [1,0,1,0,1,0]
   42
   > fromBin [1,0,1]
   5

#-}
```

This description is shown when a student starts the exercise.

The configuration file also contains a list of QuickCheck properties. QuickCheck (Claessen and Hughes, 2000) is a library for property-based testing in Haskell. The list of QuickCheck properties can be viewed as a semi-formal specification of the program. QuickCheck automatically tests properties on a large number of randomly generated test cases.

The following property is responsible for the feedback of the counterexample in the example session from the previous subsection:

```
propModel f bs = feedback msg (output == model)
  where
    output = f bs
    model  = foldl (λn b → 2 ∗ n + b) 0 bs
    msg    = "Your implementation is incorrect for the " ++
             "following input: " ++ show bs ++ "\nWe expected " ++
             show model ++ ", but we got " ++ show output
```

The property *propModel* takes a student program (f) as argument and checks that for every list of bits the output of the student program is equal to the output of a model solution (*model*). A teacher can use the *feedback* function to attach a feedback message to a property. This message will be shown when the property has been falsified.

Another property we check is that removing the most significant bit from a (non-empty) list is the same as dividing the corresponding decimal value by two. The following QuickCheck property expresses this characteristic:

```
propDivByTwo f bs = (bs /= []) ==> f (init bs) == f bs `div` 2
```

The expression before the implication ($\Longrightarrow$) makes sure we only test with non-empty binary lists.

The third and last property tests the so-called round-trip property: if we convert a list of bits to a decimal value and back to a list of bits again, then we should end up with the same list.

```
propSound f bs = feedback msg (bs == toBin (f bs))
  where
    msg = "Converting back results in a different list of bits"
```

We omit the implementation of *toBin*, which converts an decimal value to a list of bits. The properties for an exercise are grouped together in a list:

```
properties :: [([Int] → Int) → Property]
properties =
  map (λp → forAll genBin ◦ p) [propModel, propDivByTwo, propSound]
```

The elements in the *properties* list are functions that take a student program as parameter and return a QuickCheck property. Using the QuickCheck *forAll* function we let the properties use a special-purpose binary number generator (*genBin*).

If a teacher does not want to specify any properties, she can leave the list empty. This does not mean that Ask-Elle will skip testing a student program. It is always possible to construct a property that compares a student program against one of the model solutions. However, if a teacher wants to attach a specific feedback message to the property that validates against a model solution, she needs to define it herself, like we showed above.

Thus to add a programming exercise to Ask-Elle, a teacher specifies the exercise text, model solutions for the exercise, and possibly extra feedback and properties that a solution should satisfy. A teacher does not need to have any knowledge of the internals of Ask-Elle. This exemplifies our contribution that feedback and hints are derived automatically from teacher-specified annotated solutions for a problem.

3 The design of Ask-Elle

The previous section illustrated how Ask-Elle can be used for class 3 programming exercises by students and teachers. Teachers can add programming exercises to Ask-Elle by providing a task description for the exercise, one or more model solutions, and properties that a solution should satisfy. A teacher can annotate properties and model solutions with feedback messages, and can specify how much flexibility is allowed in student solutions. This section shows the design of Ask-Elle, and contains our main research contribution. It gives the architecture of Ask-Elle, and briefly discusses the technologies we use for diagnosing student programs.

Ask-Elle is a programming tutor

- that targets first year computer science students,
- in which a student incrementally develops a program that is equivalent (modulo syntactic variability) to one of the teacher-specified model solutions for a programming problem,
- that gives feedback and hints on intermediate, incomplete, and possibly buggy programs, based on teacher-specified annotations in model solutions,
- that reports counterexamples when properties for an exercise are not satisfied,
- to which teachers can easily add their own programming exercises, and in which teachers can adapt feedback,
- and in which a student can use her preferred step-size in developing a program: from making a minor modification to submitting a complete program in a single step.

Using our programming tutor students develop programs by making small, incremental, changes to a previous version of the program. Other common scenarios in teaching programming are to give a student an incomplete program, and ask her to complete the program, or to give a student a program, and ask her to change the program at a particular point. In such assignments, a student *refines* or *rewrites* a program. Both rewriting and refining preserve the semantics of a program; refining possibly makes a program more precise.

The feedback that we offer, such as giving a hint, is derived from a strategy. Strategies play a central role in our approach. We use strategies to capture the procedure of how to solve an exercise. A strategy describes which basic steps have to be taken, and how these steps are combined to arrive at a solution. In case of a functional programming exercise, the strategy describes how to incrementally

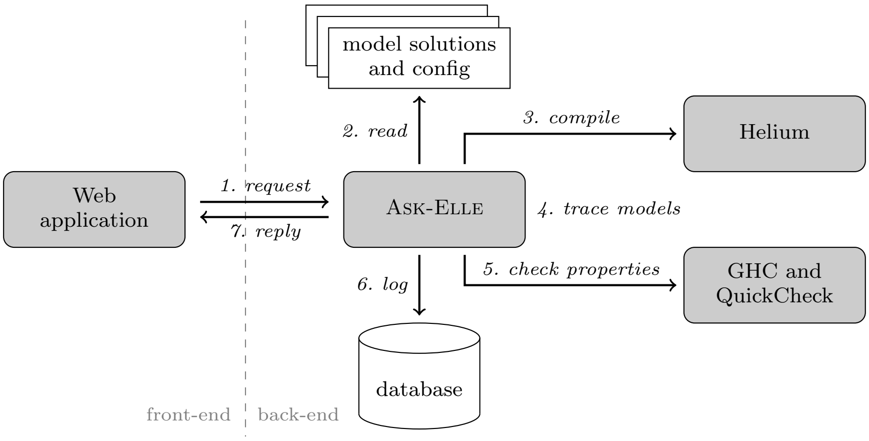


Figure 3 Ask-Elle's web-based architecture

construct a program. We reuse an embedded domain-specific language (EDSL) for defining strategies for programming (Heeren et al., 2010).

We will first present Ask-Elle's web-based architecture, and then discuss the technology behind the tutor's model tracing capabilities. We conclude this section with property checking for finding and reporting counterexamples.

3.1 Ask-Elle's web-based architecture

Our tutor can be accessed via a web browser. On the main page, a student selects an exercise to work on. While developing a program, a student can check that she is still on a path to a correct solution, ask for a single hint or all possible choices on how to proceed at a particular stage, or ask for a worked-out solution.

The programming tutor consists of a front-end and back-end. The front-end handles the interaction with the student, such as displaying feedback messages. The back-end takes care of the feedback generation. Figure 3 shows a schematic overview of our programming tutor architecture: we explain each of the steps in this figure.

The front-end of the tutor is implemented as a web application, using Ajax technology to call services. Each time a student clicks a button such as `Check` or `Hint`, our web application sends a service request *(1)* to the functional programming domain reasoner (the back-end). This request contains all information about where the student is in solving the exercise, including the current program. Requests are encoded in a simple JSON format.

The back-end of our tutor is a domain reasoner, also called the expert knowledge module in the traditional four-component ITS architecture described by Nwana (1990), for the domain of functional programming exercises in Haskell. The domain reasoner is built using a generic software framework for specifying domain reasoners (Heeren and Jeuring, 2014)[3]. This framework offers feedback

[3] `http://ideas.cs.uu.nl/www/`

services to external learning environments. The feedback services handle particular requests, and are based on the stateless client-server architecture. For example, the *diagnose* feedback service is used to analyse a student step, i.e., an intermediate program that is submitted by the student. Another example of a feedback service is *derivation*, which generates a complete worked-out example showing all steps a student can take to solve an exercise.

The domain reasoner starts with reading the model solutions and configuration file *(2)* for an exercise. These files are turned into a programming strategy (for model tracing), properties (for property-based testing), and feedback scripts (for generating feedback messages). We then compile *(3)* the student program with the Helium compiler, which is a Haskell compiler designed for reporting good error messages and warnings targeted at students learning the language. We extended the compiler to deal with the incomplete programs that can be submitted with our tutor. Compilation either produces an error message, which is reported to the student, or delivers an abstract syntax tree (AST). This AST is compared against the model solutions *(4)* using the programming strategy (details in Section 3.2). If the submitted program cannot be recognized, we check the properties *(5)* to search for counterexamples (details in Section 3.3).

At this point we use the results of steps 3–5 and the feedback script to produce a reply that is returned to the front-end *(7)*. The request-reply pair, together with some meta-information, is stored in a database *(6)* for later analysis.

3.2 Model tracing

A programming strategy is derived from the set of model solutions of an exercise. We use this strategy to track the progress of a student, and to calculate semantically rich feedback. The programming strategies are expressed in the strategy language (Heeren et al., 2010) that is supported by the general software framework on which ASK-ELLE is built. The strategy language offers combinators for choice, sequence, repetition, labels, and many more. Such a strategy can be interpreted by the framework as a context-free grammar, i.e., a set of sentences consisting of basic steps, which turns model tracing into a parsing problem.

In the case of ASK-ELLE, the basic steps are refinement rules to make an intermediate program more defined. We have defined refinement rules for most language constructs, for example, to replace a hole • by the conditional expression **if** • **then** • **else** • with three new holes. We use the strategy language to control the order in which the refinement rules are applied (e.g., the condition before the two branches, but without further restrictions – refinement steps in the two branches can even be interleaved). In the process of generating the programming strategy, we also take care of the annotations that have been specified by a teacher in the model solutions.

Programming languages typically offer all kinds of syntactical variations to write (essentially) the same program, and Haskell is no exception in this. For instance, $\lambda x \to \lambda y \to x$ and $\lambda x\ y \to x$ have a different representation, but should often be treated the same during model tracing and feedback generation. Before we compare a student (intermediate) program to the programs generated by the strategy, we *normalise* all programs using various program transformations. All transformations preserve the semantics of a program. For example, ASK-ELLE has a transformation for renaming variables (a transformation that is known as alpha-conversion in the lambda calculus (Barendregt, 1984)), removing syntactic sugar (desugaring), and algebraic properties such as $n + 1 = 1 + n$.

3.3 Property-based testing

In 2013 we added property-based testing to Ask-Elle. Properties of an exercise, such as that *sort* returns a permutation of the input list that is non-descending, are specified in its configuration file and are tested on a student program when none of the model solutions can be recognised. We use the QuickCheck library (Claessen and Hughes, 2000) for testing, which supports specifying and testing properties, and defining customized random input generators. 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 a student program.

We use GHC[4], the default compiler for Haskell, to test properties and to evaluate a student program. Again, special care is needed to deal with holes that appear in incomplete programs. When QuickCheck runs into a hole, we discard the test case and continue with the next, instead of reporting the hole as a counterexample. Because of Haskell's lazy evaluation, we can find counterexamples early on for programs with holes.

To summarize this section: we have presented Ask-Elle's web-based design that combines model tracing with property checking. We generate programming strategies from model solutions and configuration files, and these strategies are used by a generic software framework to trace the refinement steps that a student takes. We use QuickCheck for property checking, even for student programs that contain holes. With this design we can automatically calculate feedback and hints, which teachers can further adapt to their needs.

4 Experiment 1: assessing student programs

Traditionally, a teacher or a teaching assistant assesses a student's abilities and progress. However, providing timely feedback is not always possible with large class sizes. Repeatedly assessing student exercises is tedious, time consuming, and error prone. It is difficult to keep judgements consistent and fair. To assist teachers in assessing programming assignments, many assessment tools have been developed.

In the first experiment, conducted in 2009, we use Ask-Elle as an assessment tool for student programs. The assessment tool uses programming strategies and programming transformations to classify programs based on model solutions for the assignment. Using programming strategies we can guarantee that a student program is equivalent to a model solution, and we can report which solution strategy has been used to solve a programming problem. The assessment tool only assesses fully defined programs, i.e., there are no holes present in the program. The normalisation procedure exploits this fact and performs a number of program transformations that are not allowed on intermediate programs, such as removing

[4] `https://www.haskell.org/ghc/`

program code that is not used in the main functionality (dead-code elimination), and inlining.

4.1 Automated assessment with programming strategies

Many programming exercise assessment tools are based on some form of testing (Ala-Mutka, 2005). Test-based assessment tools try to determine correctness by comparing the output of a student program to the expected results on test data. Using testing for assessment has a number of problems. First, an inherent problem of testing is coverage: how do you know you have tested enough? Testing does not ensure that the student program is correct. Second, assessing design features, such as the use of good programming techniques or the absence of imperfections, is hard if not impossible with testing. This is unfortunate, because teachers want students to adopt good programming techniques. Consider the following function that solves the problem of converting a list of binary numbers to its decimal representation from the example session:

```
fromBin :: [Int] → Int
fromBin []       = 0
fromBin (b : bs) = b ∗ 2 ^ (length (b : bs) − 1) + fromBin bs
```

This function returns correct results, hence test-based assessment tools will most likely accept this as a good solution. However, the length calculation is unusual, because an element is added to the list and then the length of the list is subtracted by one. We found this imperfection frequently in a set of student solutions. Third, testing cannot reveal which algorithm has been used. For instance, when asked to implement quicksort, it is difficult to discriminate between bubblesort and quicksort. Fourth, testing is a dynamic process and is therefore vulnerable to bugs, and even malicious features, that may be present in solutions.

We use programming strategies, derived from teacher annotated model solutions, and our normalisation procedure to assess functional programming exercises in Haskell. Our approach is rather different from testing: we can guarantee that the submitted student program is equivalent to a model program. We can recognise many different equivalent solutions from a model solution. For example, the following student solution:

```
fromBin = fromBaseN 2
fromBaseN b n = fromBaseN′ b (reverse n)
  where
    fromBaseN′ _ []        = 0
    fromBaseN′ b′ (c : cs) = c + b′ ∗ (fromBaseN′ b′ cs)
```

is recognised from the *foldl* model solution (see Figure 2). Despite the fact that the student solution appears very different from the model solution, they will be recognised as equivalent. The two solutions use a similar recursion pattern and are essentially the same. In fact, the explicit recursive definition of *fromBaseN′* is recognised as an instance of the *foldr* function. In contrast to *foldl*, *foldr* combines the list elements from right to left. For example:

$$\mathit{foldl}\ (\oplus)\ e\ [x_1, x_2, x_3] = ((e \oplus x_1) \oplus x_2) \oplus x_3$$
$$\mathit{foldr}\ (\oplus)\ e\ [x_1, x_2, x_3] = x_1 \oplus (x_2 \oplus (x_3 \oplus e))$$

The *foldl* function can be defined as a *foldr* by reversing the list and flipping the operator's arguments. We specify this as an `ALT` annotation:

`{-# ALT` $\mathit{foldl}\ \mathit{op}\ b = \mathit{foldr}\ (\mathit{flip}\ \mathit{op})\ b \circ \mathit{reverse}$ `#-}`

With this annotation, the student solution (with explicit recursion) and the model solution (with *foldl*) are normalized to the same program.

In the remainder of this section we show how programming strategies and program transformations can be used to assess functional programming exercises. Using strategies for assessing student programs solves the four problems of using testing for assessment described above:

1. if a program is determined to be equivalent, it is guaranteed to be correct
2. we can recognise and report imperfections
3. we can determine which algorithm has been implemented
4. strategy-based assessment is carried out *statically.*

In contrast with our strategy-based assessment approach, test-based assessment tools can give a judgement of all programs including incorrect ones. Test-based assessment tools can prove a program to be incorrect by providing a counter-example. By adding testing with QuickCheck to our assessment tool we get around this disadvantage. With this addition, however, we do not solve the fourth problem we mentioned, and become vulnerable to bugs and malicious software as well.

4.2 Programming exercise and model solutions

We have applied our assessment tool to student solutions that were obtained from a lab assignment in a first-year functional programming course at Utrecht University (2008). We were not involved in any aspect of the assignment, and received the solutions after they had been graded 'by hand' by the teaching assistants. In total we received 94 student solutions.

The students had to implement the *fromBin* function. This is a typical beginner's exercise in Haskell. The *fromBin* exercise can be solved in various ways, using different kinds of higher-order functions. We use the model solutions in Figure 2. All three model solutions are elegant, efficient, and use recommended programming techniques. We added a fourth model solution that is simple, but inefficient:

$$\begin{aligned} &\mathit{fromBin}\ [\,] &&= 0 \\ &\mathit{fromBin}\ (b : bs) &&= b * 2\,\hat{}\ \mathit{length}\ bs + \mathit{fromBin}\ bs \end{aligned}$$

This definition is similar to model solution 3 with tupling, except that the length of the list is calculated in each recursive call. Hence, it takes time quadratic in the size of the input list to calculate its result. The other model solutions are all linear. It is up to the teacher to decide to either accept or reject solutions based on this model. This flexibility is one of the advantages of our approach. The teacher can accept the solution and add feedback, which is given to the student after she completes the exercise, explaining the inefficiency.

4.3 Classification of student solutions

The most important features we want to assess in a student program are:

- Correctness: does the program implement the requirements?
- Design: has the program been implemented following good programming practices?

We have partitioned the set of student programs into four categories by hand:

Good. A good program is a proper solution with respect to the features we assess (correctness and design). It should ideally be recognised as equivalent to one of the model solutions.

Good with modifications. Some students have augmented their solution with sanity checks. For example, they check that the input is a list of zeroes and ones. Since the exercise assumes the input has the correct form, we have not incorporated such checks in the model solutions. The transformation machinery cannot yet remove such checks: we have removed them by hand.

Imperfect. We reject programs containing imperfections. The solution to *fromBin* given at the beginning of Section 4.1 is an example of an imperfect solution. Another common imperfection we found is the use of a superfluous case:

```
fromBin []       = 0
fromBin [b]      = b
fromBin (b : bs) = b * 2 ^ (length bs) + fromBin bs
```

In this example, the second case is unnecessary.

Incorrect. A few student programs were incorrect. They all contained the same error: no definition of *fromBin* on the empty list.

4.4 Results

From the 94 student programs, 64 programs fall into the good category and 8 fall into the good with modifications category. From these 72 programs, our assessment tool recognises 62 programs (86%). Another, and perhaps better, way of looking at these numbers is that 62 student solutions are accepted based on just four model solutions. All of the incorrect and imperfect programs were not recognised by our tool in the experiment, that is, we did not have any false positives. Some of these incorrect programs were not noticed by the teaching assistants that corrected these programs.

Using our tool a teacher only needs to assess the remaining student solutions. If our tool cannot recognise a program as an instance of a model solution, we can use testing to find a counterexample, such as the empty list for the incorrect student programs.

It may happen that a correct student solution does not correspond to a model solution. If such a solution is elegant and efficient, a teacher could add it to the set of model solutions. In the case it does not meet the requirements for a model solution, it is up to the teacher to take a decision. For example, the following student solution uses the tupling technique:

$$
\begin{aligned}
&\mathit{fromBin}\ [\,] &&= 0\\
&\mathit{fromBin}\ [b] &&= b\\
&\mathit{fromBin}\ (b : c : \mathit{rest}) &&= \mathit{fromBin}\ ((2 * b + c) : \mathit{rest})
\end{aligned}
$$

Instead of using a tuple or an extra argument, this solution 'misuses' the head of the list to store the result. The teacher needs to decide whether or not this misuse is an imperfection or not.

By checking all model solutions independently, we can tell which model solution, or strategy, a student has used to solve the exercise. Our test showed that 18 students used the *foldl* model solution, 2 used tupling, 2 the inner product solution, and 40 solutions were based on the extra model solution with explicit recursion.

It is unlikely that a solution is accepted by more than one model solution. In our test all solutions were accepted by a single model solution. If model solutions are very similar, it might be possible to use an `ALT` annotation to recognise both from a single model solution.

4.5 Related work on assessment

The survey of automated programming assessment by Ala-Mutka (2005) shows that many assessment tools are based on dynamic testing. In contrast, our assessment tool statically checks for correctness. The survey provides many pointers to related work. We describe the three closest approaches.

The PASS system, developed by Thorburn and Rowe (1997), assesses C programs by evaluating whether a student program conforms to a predefined solution plan. A drawback of the system is that it needs testing for this evaluation. Moreover, a solution plan is much more strict compared to a strategy. For example, the system considers the definition of any helper function incorrect. Our approach allows a higher degree of freedom by means of standard strategies and program transformations.

The approach of Truong et al. (2004) is also based on model solutions and abstract syntax tree inspections. However, their primary use is to assess software quality and not so much correctness. In addition to similarity checks, their system also calculates software metrics, which are used to give feedback to a student. A drawback of their approach is that it does not take the different syntactic forms of a model solution into account. Moreover, the similarity check considers only the outline of a solution and not its details.

Xu and Chee (2003) show how to diagnose Smalltalk programs using program transformations. Their approach is rather similar to our approach. The set of transformations for a functional programming language is much smaller and simpler. We would like to implement their advanced method for locating errors in student programs.

Program verification tools are used to prove programs correct with respect to some specification (Mol et al., 2002). Automatic program verification tools provide as much support as possible in constructing this proof. However, users always need to give hints or proof steps to complete proofs for non-trivial programs, such as *fromBin*.

#	Question	Score
1	The tutor helped me to understand how to write simple functional programs	3.15
2	I found the high-level hints about how to solve a programming problem useful	3.43
3	I found the hints about the next step to take useful	3.05
4	The step-size of the tutor corresponded to my intuition	2.85
5	I found the possibility to see the complete solution useful	4.25
6	The worked-out solutions helped me to understand how to construct programs	3.55
7	The feedback texts are easy to understand	3.25
8	The kind of exercises offered are suitable for a first functional programming course	3.90

Table 1 Questionnaire: questions and scores.

5 Experiment 2: questionnaire

We have used our functional programming tutor in a course on functional programming for bachelor students at Utrecht University in September 2011. The course attracted more than 200 students. Around a hundred of these students have used our tutor in two sessions in the second week of the course after three lectures. 40 students filled out a questionnaire about the tutor, and we collected remarks during the lab sessions. Table 1 shows the questions and the average score on a Likert scale from 1 to 5. The first seven questions are related and indicate how satisfied students are with the tutor. The last question addresses how students value the difficulty of the offered exercises.

At the time of this experiment, there was no support for property-based testing in the Ask-Elle tutor. The goal of this experiment was to analyse if students appreciate our incremental approach, such as giving feedback on intermediate answers, and to discover which parts of the tutor need further improvements.

5.1 Reflection on the scores

The scores show that the students particularly like the worked-out solution feedback. A worked-out solution presents a complete, step-wise, construction of a program. Furthermore, the kind of exercises are as expected by the students. The results also show that the step-size used by the tutor does not correspond to the intuition of the student. We also noticed this during the experiment. The students often took larger steps than that the tutor was able to handle.

The average of the first seven question gives an overall score of the tutor of 3.4 out of 5. This is sufficient, but there is clearly room for improvement.

5.2 Evaluation of open questions

In addition to questions about the usage of the tutor, the questionnaire also contained some general questions, including:

1. We offer the feedback services: *strategy hint*, *step hint*, *step*, *all steps*, *solution*, and we check the program submitted by the student. Do you think we should offer more or different feedback services?

2. Do you have any other remarks, concerns, or ideas about our programming tutor?

The answers from the students to the first question indicate that the feedback services are adequate. We received some interesting suggestions on how to improve our tutor in response to the second open question. The remarks that appear most are:

- Some solutions are not recognised by the tutor
- The response of the tutor is sometimes too slow

The first remark may indicate that a student believes her own solution is correct, where in fact it is not. It could be that the program is incorrect or contains imperfections, such as being inefficient, and hence is rejected by the tutor. This remark addresses the fact that we cannot give feedback on student programs that deviate from a path towards one of the model solutions. When a student program deviates from a path towards a model solution there are three possibilities:

- The student program is incorrect, and we should be able to detect this and give a counterexample. This observation has resulted in adding property-based testing to Ask-Elle (see Section 3.3).
- The student program is correct and uses desirable programming techniques, but our tutor rejects it. In this case the set of model solutions should be extended with this solution.
- The student program is functionally correct but contains some imperfections, such as $length\ (x : xs) - 1$ which should be simplified to $length\ xs$. In general, the tutor cannot conclude that a student program contains imperfections when it passes the tests but deviates from the strategy, and therefore it cannot give a definitive judgement. However, after using an exercise in the tutor for a while, and updating the tutor whenever we find an improvement, it is likely that the set of model solutions is complete, and therefore unlikely that a student comes up with a new model solution (see experiment 1 on the assessment of student programs). Therefore, in this particular case we can give feedback that a student program probably has some undesired properties.

The second remark is related to the step-size supported by the tutor. When a student takes a large step, the tutor has to check many possibilities, due to the flexibility that our tutor offers. We have addressed this problem by constraining the search space and by introducing a special search mode that is used for recognising steps. The technical details can be found in (Gerdes et al., 2012a) and its accompanying technical report.

In addition to the described experiment, we also asked a number of functional programming experts from the IFIP WG 2.1 group[5] and student participants of the Central European Functional Programming (CEFP 2011) summer school to fill out a questionnaire. We asked for input about some of the design choices we made in our tutor, such as giving hints in three levels of increasing specificity. Both the experts as well as the students support most of the choices we made. The main suggestion we got for adding extra services/functionality was to give concrete counterexamples using testing for semantically incorrect solutions.

[5] http://foswiki.cs.uu.nl/foswiki/IFIP21/

6 Classification for student programs

Based on the observations from the two experiments, we can now present a detailed classification for student programs that takes correctness and design into account. 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 and GHC to compile student programs. Both compilers report syntax and type errors, which the student first has to repair.

Matches model solution (Model). Ask-Elle can match the 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 undefined at too many places.

The first three categories correspond to steps 3–5 in Figure 3. Ideally, the number of programs in the *Undecided* 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 testing with QuickCheck sometimes cannot find counterexamples for incorrect programs. We give answers to the above questions at the end of Section 7.

6.1 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 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 do not check such style issues in Ask-Elle, although we might use a tool such as HLint[6] 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.

We briefly reflect on categorising the 94 student programs from the assessment experiment described in Section 4. The 62 recognized student solutions are classified as *Model*. However, the sanity checks in the 8 *good with modifications* programs cannot be removed automatically, and are thus classified as *Tests passed*; this is an example of case (3). The *imperfect* student programs also end up in *Tests passed* and illustrate case (4). A counterexample is found for all the incorrect programs. Because we only collected final programs for this experiment, none of the programs is classified as *Error* or *Discarded*.

6.2 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.

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

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.

7 Experiment 3: student program analysis

In the third experiment we 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 *fromBin*, *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.

The 3466 interactions with Ask-Elle come from 116 out of the 285 students registered for the course. Students seem to have worked top-down through the list of exercises: the exercises *dupli*, *range*, and *repli* have been tried 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 into the categories given in the previous section. To classify an attempt we use its last interaction. The overall results are shown in Table 2. The results for the functions for which we received the most interactions (*dupli*, *repli*, and *compress*) are shown in Table 3.

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	
recognised	221/477	(46.3%)	754/1345	(56.1%)
classified	396/652	(60.7%)	2875/3466	(82.9%)

Table 2 Categorising student programs

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:

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

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.

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% are classified as correct or incorrect. Of all attempts, 60.7% are 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 *dupli* $\bullet = \bullet$ is classified as *Model*.
- A possible reason for why the results for *dupli* are better than the results for the other two exercises is that there was a bug in the renaming of variables, which was found during and repaired after the experiment. This bug would fire sooner in definitions with two parameters instead of one. Another reason can be 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 analysed the set of programs in *Tests passed* and *Discarded* to discover which program transformations or model solutions we should add. The results are discussed in Section 7.1. The *recognised* ratio can also be increased by improving the properties that are used to find counterexamples. During the analysis we found that some properties needed

Exercise	Category	Attempts		Interactions	
dupli	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	
	recognised	65/93	(69.9%)	184/261	(70.5%)
	classified	113/141	(80.1 %)	719/796	(90.3%)
repli	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	
	recognised	12/47	(25.5%)	40/119	(33.6%)
	classified	30/65	(46.2%)	330/409	(80.7%)
compress	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	
	recognised	11/38	(28.9%)	104/183	(56.8%)
	classified	34/61	(55.7%)	400/479	(83.5%)

Table 3 Categorising student programs for *dupli*, *repli*, and *compress*

adjustments. For example, changing the test case generators can result in discarding fewer test cases and finding more counterexamples. This would decrease the number of programs in the *Tests passed* and *Discarded* categories.

7.1 Program transformation for student programs

For all 436 student programs of all exercises in the *Tests passed* category, we determined by hand whether or not they can be recognised if we would add or improve Ask-Elle's program transformations. We collected a list of program transformations that could 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.

Adding model solutions to the tutor is very simple and requires no programming: this can always be done for solutions that are not recognized. Improving the tutor's program transformations, however, takes a lot of effort. Semantic equality of programs is undecidable, and there does not exist a normal form for programs.

This implies that there is no complete set of transformations to use when transforming student programs and model solutions. In our analysis we have been conservative with marking programs as new model solutions, because the goals is to improve Ask-Elle's matching capabilities. Admittedly, the distinction between a new solution and a program that can be recognized from an existing solution is not sharp and requires judgement from a teacher or expert.

Below we list the program transformations that have to be added or improved to match a student program to a model solution. These program transformations are standard techniques for normalising and optimising functional programs that are often found in compilers. These transformations are general and not specific for the exercises we used. Note that one 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 (performing the application of a function to an argument (Barendregt, 1984)), helps in 39 cases.
3. Dealing with function parameters uniformly. For example, the student program $\mathit{palindrome} = (\lambda x \rightarrow x == \mathit{reverse}\ x)$ with a lambda-expression is not matched against the model solution $\mathit{palindrome}\ x = x == \mathit{reverse}\ x$, although these definitions are equivalent. Not using function composition $(\circ)$ is another example, e.g. $\mathit{dupli}\ x = \mathit{concatMap}\ (\mathit{replicate}\ 2)\ x$ versus model solution $\mathit{dupli} = \mathit{concatMap} \circ \mathit{replicate}\ 2$. Often, some form of eta-conversion (abstracting from an argument, removing a lambda, or, the other way around, introducing a lambda (Barendregt, 1984)), for example, replacing $(\lambda x \rightarrow (+)\ 1\ x)$ by $(+)\ 1$, is sufficient to match more solutions. There are $8 + 54 + 13 = 75$ occurrences of these transformations. We expect such programs can be recognised by introducing eta-conversion, and by normalising definitions with parameters to lambda-expressions (such as for the *palindrome* example).
4. The alpha-conversion 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, removing syntactic sugar from the program would help, such as converting the Haskell list-notation $[1, 2, 3]$ to constructor application $1{:}2{:}3{:}[\,]$.
8. One program requires removing an unused (helper) definition. We cannot remove an unused helper function in an incomplete program with holes, because such a definition may still be used 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 function bindings, transforming between guards, patterns, and if-then-else conditionals, etc.

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. We give an example in Section 7.3.

7.2 Results

We return to the questions posed in Section 6.

- How many programs are classified as undecided? 17.1% of all interactions and 39.3% 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? Consider the 436 programs (interactions) that were in the *Tests passed* category. By adding new transformations, Ask-Elle now recognises 161 of these programs as model solutions. By fixing the alpha-conversion 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 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 instead of pattern matching directly on a list, to get access to the first element and the rest of the list, they use the prelude functions *head* and *tail*.
- 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

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	
recognised	1095/1340	(81.7%)
classified	3221/3466	(92.9%)

Table 4 Categorising student programs with improved program transformations and new model solutions

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 $dupli\ xs = map \bullet xs$. This definition is correct for the input $[\,]$, but will be incorrect for any non-empty input. However, all tests with non-empty inputs are discarded. This error may be caught by considering additional properties about the length of a duplicated list.

We give the version of Table 2 for interactions, taking the new and improved program transformations and new model solutions into account, in Table 4. 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%).

7.3 Abstract model solutions

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:

$$
\begin{array}{lll}
range_1\ a\ b & \mid a \mathrel{==} b & = [a] \\
 & \mid otherwise & = a : range_1\ (a+1)\ b \\
range_2\ a\ b & \mid a \mathrel{==} b & = [b] \\
 & \mid otherwise & = a : range_2\ (a+1)\ b \\
range_3\ a\ b & \mid a > b & = [\,] \\
 & \mid otherwise & = a : range_3\ (a+1)\ b \\
range_4\ a\ b & \mid a > b & = [\,] \\
 & \mid otherwise & = range_4\ a\ (b-1) \mathbin{+\!\!+} [b]
\end{array}
$$

$$\begin{array}{llll} \mathit{range}_5\ a\ b & \mid a \leqslant b & = a : \mathit{range}_5\ (a+1)\ b \\ & \mid \mathit{otherwise} & = [\,] \end{array}$$

The first definition can be transformed in the second definition by means of program transformation 6 (from Section 7.1). 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:

$$\begin{array}{l} \mathit{condIterate}\ \mathit{cond}\ \mathit{begin}\ \mathit{it}\ \mathit{af}\ \mathit{bf} = \mathit{step} \\ \quad \textbf{where} \\ \qquad \mathit{step}\ a\ b \\ \qquad\quad \mid \mathit{cond}\ a\ b \;\; = \mathit{begin} \\ \qquad\quad \mid \mathit{otherwise} = \mathit{it}\ a\ b\ (\mathit{step}\ (\mathit{af}\ a)\ (\mathit{bf}\ b)) \\ \mathit{range}_3 = \mathit{condIterate}\ (>)\ [\,]\ (\lambda a\ \_\ \mathit{xs} \to a : \mathit{xs})\ (+1)\ \mathit{id} \end{array}$$

We have yet to investigate laws for abstract functions such as *condIterate*, and the kind of program transformations necessary to use this approach.

8 Related work

This section describes the related work in the area of intelligent tutoring for learning programming: related work on assessment has been discussed in Section 4 on the assessment experiment we performed. Through the years several hundreds of papers on intelligent tutoring systems and learning environments for learning programming have been published. This section can only discuss a fraction of those. We provide references to review papers in which interested readers can find more information about various aspects of tutoring systems for learning programming.

Tools that support students in learning programming have been developed since the 1960s (Ulloa, 1980; Douce et al., 2005). Some of these tools analyse incremental steps a student takes (Anderson et al., 1986), and/or support the development of programming plans (Soloway, 1985; Johnson and Soloway, 1985). The early work in this area primarily targeted the programming languages Lisp and Pascal (including Pascal-like imperative languages); a nice overview is given by Vanneste et al. (1996).

Our work is probably closest to the Lisp tutor and its successors (Anderson et al., 1986; Brusilovsky et al., 1996; Corbett and Anderson, 2001). The main difference with these tutors is that Ask-Elle offers class 3 exercises, and that adding a programming exercise to Ask-Elle is relatively easy, and only requires authors to provide annotated model solutions and properties in Haskell.

The next wave of programming tutors primarily targeted Java (Kölling et al., 2003; Sykes and Franek, 2004; Holland et al., 2009) and Prolog (Sison et al., 2000; Hong, 2004; Le et al., 2009), but tutors for other programming languages have been developed as well, of course. A recent overview by Le et al. (2013) describes the various AI-supported approaches that have been used in programming tutors.

Striewe and Goedicke (2014) review one such approach in detail: static analysis for diagnosing student programs.

A current trend in the intelligent tutoring systems world is to use data-driven techniques to give feedback and hints to users of intelligent tutoring systems and other learning environments (Rivers and Koedinger, 2014; Jin et al., 2012; Price and Barnes, 2015). Feedback and hints are now generated from previous student solutions to a programming exercise, and possibly a seeding expert solution, instead of from a complete collection of model solutions for a program. The techniques used by Rivers and Koedinger (2013) are close to some of the techniques we use: student solutions to a programming exercise are represented in a solution space obtained by applying program transformations to student programs to obtain programs in some normal form. This approach is particularly useful when it is hard to come up with a more or less complete set of model solutions. A possible disadvantage is the reduced teacher control over feedback and hints.

Many of the techniques we use have been used in earlier tutors, but none of these early or more recent tools combines strategy-based model tracing and property-based testing to construct a programming tutor for class 3 exercises, in which students get automatic feedback and hints on intermediate steps, and in which teachers can easily add exercises and adapt feedback.

As far as we are aware, the kind of analysis performed in Section 7 has not been performed before. Designers of tutoring systems for learning Prolog (Johnson, 1990; Looi, 1991; Hong, 2004; Le and Menzel, 2009) have analysed complete student programs in a similar fashion, but we think also analysing intermediate diagnoses of student programs is essential for determining the quality of an intelligent tutoring system. There has been quite some work on semantic-preserving variations (Xu and Chee, 2003; Wang et al., 2007), used to equate student programs with model solutions. These variations correspond to our program transformations. The kind of program transformations we apply differ significantly from earlier work, because we have a Haskell tutor instead of a tutor for imperative programming.

9 Conclusions and future work

We have shown the design of Ask-Elle, a tutor for the lazy, typed, higher-order functional programming language Haskell. We have performed several experiments with the tutor, and have shown how these experiments influenced the design of the tutor. Ask-Elle supports the incremental development of programs for class 3 programming exercises, offering feedback and hints while a student is developing a program. The feedback and hints are automatically calculated from teacher-specified annotated model solutions and from properties that a solution to the exercise should satisfy. The main technologies we use to provide feedback and hints are strategy-based model tracing and property-based testing.

Our experiments focus on determining the quality of the diagnoses of Ask-Elle. Before we can use Ask-Elle to measure learning effects or learning efficiency, we first need to make sure that the feedback is accurate and perceived as useful. We performed several experiments in which we tested the accuracy and the perceived usefulness of the analyses. We expect that with the implementation of the improvements obtained from the analysis in our last experiment, the feedback of Ask-Elle is good enough to use it in the daily practice of a course. Taking the improvements found in the last experiment into account, we are able to recognise nearly 82% and classify nearly 93% of all interactions. We will use this version of Ask-Elle to measure learning effects and effectiveness.

The second experiment shows that students highly value the worked-out examples from Ask-Elle. This is in line with three decades of experimental research on learning by observing and imitating examples in which an expert demonstrates how to solve a problem versus learning by doing. For novices, who lack prior knowledge of a task, observing examples or alternating example-study and problem solving is more effective and efficient (leads to better learning outcomes in less time and with less effort) compared to practising problem solving (Sweller et al., 2011). We advise teachers to use Ask-Elle first to study the incremental development of programs for some example exercises.

Ask-Elle is particularly useful for beginner's exercises in Haskell, such as exercises in which a student has to fill out a missing component, or develop a small program performing a particular task. For such exercises, it usually suffices to construct about five model solutions that cover the range of solutions students submit. We expect that the hints of Ask-Elle help beginners realise that there are different ways to solve a single problem. Ask-Elle is less suited for larger projects, or for programs that involve I/O, since for these kinds of exercises the amount of possible solutions gets very large, and specifying all possible model solutions becomes impractical. This is not a problem: we think the kind of feedback and hints provided by Ask-Elle are very useful for a beginner, and less useful for a more advanced Haskell programmer. For example, a teacher might use Ask-Elle in the first part of a course, say the first 50-100 hours spent by a student, and then use the standard compiler for Haskell, and provide feedback and hints in a different way, or on a different level. To promote deep learning, we think debriefing sessions after using Ask-Elle during a session or some period of time are essential for student learning.

In this paper we described a tutor for learning the functional programming language Haskell. We believe, however, that our approach based on programming strategies is also applicable to other programming languages and programming paradigms, because the concepts on which our approach is based, such as strategies, refinement rules, and program transformations, are applicable to every programming language. A prototype of a tutor for introductory imperative programming problems that is based on the same technologies was recently presented by Keuning et al. (2014).

Ask-Elle shows that to extend a programming tutor with a new exercise, to which a teacher wants to add a particular kind of feedback, a teacher only needs to construct model solutions with feedback annotations, and does not need to know anything about the internals of a tutoring system. This significantly reduces the burden to use such a tutor.

9.1 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 a 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. 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.

If a student program is syntactically incorrect, or if it contains a type error, we cannot give any hints or feedback based on model solutions. Instead, in these situations we rely on the Helium compiler for error messages. We want to investigate if we can use error-correcting parsing and other compiler techniques to also give strategy-based model tracing hints and feedback on syntactically incorrect or type-incorrect programs. We expect that with a good error-correcting parser (Swierstra and Alcocer, 1999) we can deal with quite a few syntax errors, and we might be able to provide more feedback and hints for a significant amount of the 55% of student programs that contain compiler errors. We are less sure if a similar approach would work for programs with type errors.

If a student program is incorrect, we report a counterexample. A counterexample does not tell a student where the error occurs in the program. We want to make use of contracts (Meyer, 1992) and types to propagate properties of solutions to the components of a student program, to more precisely point to the part of a program that violates a desired property.

We want to investigate if we can specify model solutions to programming exercises as much as possible with abstract functions such as *condIterate* (see Section 7.3) and variants of folds, and if we can use laws for these functions and program transformations to reduce the number of model solutions, and to increase the number of recognised programs.

We are also currently performing a systematic literature review on what kind of feedback is generated in learning environments for programming, what kind of techniques are used to generate feedback, and how adaptable feedback is. We hope to use the results of this review to further develop Ask-Elle.

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful suggestions. We also thank Andres Löh, Doaitse Swierstra, and Jurriaan Hage for allowing us to perform experiments with our tutor in their classes. For the assessment experiment, Stefan Holdermans provided a set of suitable programming exercises, accompanied with a large amount of corrected student solutions. Bram Vaessen analysed the scores of our questionnaire. Peter van de Werken, Bram Schuur, Tom Tervoort and Gabe Dijkstra contributed to the source code of Ask-Elle. Hieke Keuning found some of the related work, and is working on a systematic literature review on automated feedback for programming exercises.

References

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102.

Anderson, J. R., Conrad, F. G., and Corbett, A. T. (1986). Skill acquisition and the LISP tutor. *Cognitive Science*, 13:467–505.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. (1995). Cognitive tutors: lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207.

Barendregt, H. P. (1984). *The Lambda Calculus: its Syntax and Semantics, revised edition*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

Bird, R. S. (1998). *Introduction to functional programming using Haskell*. Prentice-Hall.

Bokhove, C. and Drijvers, P. (2010). Digital Tools for Algebra Education: Criteria and Evaluation. *International Journal of Computers for Mathematical Learning*, 15(1):45–62.

Brusilovsky, P., Schwarz, E., and Weber, G. (1996). Elm-art: An intelligent tutoring system on world wide web. In Frasson, C., Gauthier, G., and Lesgold, A., editors, *Intelligent Tutoring Systems*, volume 1086 of *LNCS*, pages 261–269. Springer-Verlag.

Claessen, K. and Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP 2000: the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM.

Corbett, A., Anderson, J., and Patterson, E. (1988). Problem compilation and tutoring flexibility in the LISP tutor. In *Intelligent Tutoring Systems*, pages 423–429.

Corbett, A. T. and Anderson, J. R. (2001). Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of CHI '01: the SIGCHI Conference on Human Factors in Computing Systems*, pages 245–252. ACM.

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3).

Gerdes, A., Heeren, B., and Jeuring, J. (2012a). Teachers and students in charge — Using annotated model solutions in a functional programming tutor. In *Proceedings EC-TEL 2012: the 7th European Conference of Technology Enhanced Learning*, volume 7563 of *LNCS*, pages 383–388. Springer-Verlag.

Gerdes, A., Jeuring, J., and Heeren, B. (2010). Using strategies for assessment of programming exercises. In Lewandowski, G., Wolfman, S. A., Cortina, T. J., and Walker, E. L., editors, *Proceedings of SIGCSE 2010: the 41st ACM technical symposium on Computer science education*, pages 441–445. ACM.

Gerdes, A., Jeuring, J., and Heeren, B. (2012b). An interactive functional programming tutor. In Lapidot, T., Gal-Ezer, J., Caspersen, M., and Hazzan, O., editors, *Proceedings of ITICSE 2012: the 17th Annual Conference on Innovation and Technology in Computer Science Education*, pages 250–255. ACM.

Heeren, B. and Jeuring, J. (2014). Feedback services for stepwise exercises. *Science of Computer Programming, Special Issue on Software Development Concerns in the e-Learning Domain*, 88:110–129.

Heeren, B., Jeuring, J., and Gerdes, A. (2010). Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370.

Heeren, B., Leijen, D., and IJzendoorn, A. v. (2003). Helium, for learning Haskell. In *Proceedings of Haskell 2003: the ACM SIGPLAN Workshop on Haskell*, pages 62–71. ACM.

Holland, J., Mitrovic, A., and Martin, B. (2009). J-Latte: a constraint-based tutor for Java. In *Proceedings of ICCE 2009: the 17th International Conference on Computers in Education*, pages 142–146.

Hong, J. (2004). Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal on Human-Computer Studies*, 61(4):505–534.

Hudak, P. (2000). *The Haskell school of expression: learning functional programming through multimedia.* Cambridge University Press.

Hutton, G. (2007). *Programming in Haskell.* Cambridge University Press.

Jeuring, J., Binsbergen, L. T. v., Gerdes, A., and Heeren, B. (2014). Model solutions and properties for diagnosing student programs in Ask-Elle. In Barendsen, E. and Dagienė, V., editors, *Proceedings of CSERC 2014: Computer Science Education Research Conference*, pages 31–40. ACM.

Jeuring, J., Gerdes, A., and Heeren, B. (2012). Ask-Elle: A Haskell tutor — Demonstration —. In *Proceedings EC-TEL 2012: the 7th European Conference of Technology Enhanced Learning*, volume 7563 of *LNCS*, pages 453–458. Springer-Verlag.

Jin, W., Barnes, T., Stamper, J. C., Eagle, M. J., Johnson, M., and Lehmann, L. (2012). Program representation for automatic hint generation for a data-driven novice programming tutor. In Cerri, S. A., Clancey, W. J., Papadourakis, G., and Panourgia, K., editors, *Proceedings of ITS 2012: the 11th International Conference on Intelligent Tutoring Systems*, volume 7315 of *LNCS*, pages 304–309. Springer-Verlag.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence*, 42(1):51–97.

Johnson, W. L. and Soloway, E. (1985). Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275.

Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) and IEEE Computer Society (2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* ACM.

Keuning, H., Heeren, B., and Jeuring, J. (2014). Strategy-based feedback in a programming tutor. In Barendsen, E. and Dagienė, V., editors, *Proceedings of CSERC 2014: Computer Science Education Research Conference*, pages 43–54. ACM.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268.

Kumar, A. N. (2008). The effect of using problem-solving software tutors on the self-confidence of female students. In *Proceedings of SIGCSE 2008: the 39th SIGCSE technical symposium on Computer science education*, pages 523–527. ACM.

Le, N.-T. and Menzel, W. (2009). Using weighted constraints to diagnose errors in logic programming - the case of an ill-defined domain. *International Journal of Artificial Intelligence in Education*, 19:381–400.

Le, N.-T., Menzel, W., and Pinkwart, N. (2009). Evaluation of a constraint-based homework assistance system for logic programming. In *Proceedings of ICCE 2009: the 17th International Conference on Computers in Education*, pages 51–58.

Le, N.-T. and Pinkwart, N. (2014). Towards a classification for programming exercises. In *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science.*

Le, N.-T., Strickroth, S., Gross, S., and Pinkwart, N. (2013). A review of AI-supported tutoring approaches for learning programming. In *Advanced Computational Methods for Knowledge Engineering*, volume 479 of *Studies in Computational Intelligence*, pages 267–279. Springer-Verlag.

Looi, C.-K. (1991). Automatic debugging of Prolog programs in a Prolog intelligent tutoring system. *Instructional Science*, 20:215–263.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Mol, M. d., Eekelen, M. v., and Plasmeijer, R. (2002). Theorem proving for functional programmers – Sparkle: a functional theorem prover. In *Proceedings of IFL 2001, 13th International Workshop onImplementation of Functional Languages*, volume 2312 of *LNCS*, pages 55–72. Springer-Verlag.

Mory, E. (2003). Feedback research revisited. In Jonassen, D., editor, *Handbook of research for educational communications and technology*, pages 745–783.

Nwana, H. S. (1990). Intelligent tutoring systems: an overview. *Artificial Intelligence Review*, 4(4):251–277.

Odekirk-Hash, E. and Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. In *Proceedings of SIGCSE 2001: the 32nd SIGCSE technical symposium on Computer Science Education*, pages 55–59. ACM.

Peyton Jones, S. (2003). *Haskell 98, Language and Libraries. The Revised Report.* Cambridge University Press.

Price, T. W. and Barnes, T. (2015). Creating data-driven feedback for novices in goal-driven programming projects. In Conati, C., Heffernan, N., Mitrovic, A., and Verdejo, M. F., editors, *Artificial Intelligence in Education*, volume 9112 of *LNCS*, pages 856–859. Springer-Verlag.

Rivers, K. and Koedinger, K. R. (2013). Automatic generation of programming feedback: A data-driven approach. In *Proceedings of AIEDCS 2013: the First Workshop on AI-supported Education for Computer Science.*

Rivers, K. and Koedinger, K. R. (2014). Automating hint generation with solution space path construction. In Trausan-Matu, S., Boyer, K. E., Crosby, M., and Panourgia, K., editors, *Proceedings of ITS 2014: the 12th International Conference on Intelligent Tutoring Systems*, volume 8474 of *LNCS*, pages 329–339. Springer-Verlag.

Sison, R. C., Numao, M., and Shimura, M. (2000). Multistrategy discovery and detection of novice programmer errors. *Machine Learning*, 38:157–180.

Soloway, E. (1985). From problems to programs via plans: the content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research*, 1(2):157–172.

Striewe, M. and Goedicke, M. (2014). A review of static analysis approaches for programming exercises. In *Computer Assisted Assessment. Research into E-Assessment*, volume 439 of *Communications in Computer and Information Science*, pages 100–113. Springer-Verlag.

Sweller, J., Ayres, P., and Kalyuga, S. (2011). *Cognitive load theory.* Springer-Verlag.

Swierstra, S. D. and Alcocer, P. R. A. (1999). Fast, error correcting parser combinators: A short tutorial. In *SOFSEM99: Theory and Practice of Informatics*, pages 112–131. Springer-Verlag.

Sykes, E. and Franek, F. (2004). A prototype for an intelligent tutoring system for students learning to program in Java. *Advanced Technology for Learning*, 1(1).

Thompson, S. (2011). *Haskell: The Craft of Functional Programming*. Addison-Wesley.

Thorburn, G. and Rowe, G. (1997). Pass: an automated system for program assessment. *Computers & Education*, 29(4):195–206.

Truong, N., Roe, P., and Bancroft, P. (2004). Static analysis of students' Java programs. In *Proceedings of ACE '04: the sixth conference on Australasian computing education*, pages 317–325. Australian Computer Society, Inc.

Ulloa, M. (1980). Teaching and learning computer programming: a survey of student problems, teaching methods, and automated instructional tools. *ACM SIGCSE Bulletin*, 12(2):48–64.

VanLehn, K. (2006). The behavior of tutoring systems. *International Journal on Artificial Intelligence in Education*, 16(3):227–265.

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

Vanneste, P., Bertels, K., and Decker, B. d. (1996). The use of reverse engineering to analyse student computer programs. *Instructional Science*, 24:197–221.

Voeten, J. (2001). On the fundamental limitations of transformational design. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):533–552.

Wang, T., Su, X., Ma, P., Wang, Y., and Wang, K. (2011). Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, 56(1):220–226.

Wang, T., Su, X., Wang, Y., and Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99–107.

Xu, S. and Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384.