An Interactive Functional Programming Tutor

Alex Gerdes
School of Computer Science,
Open Universiteit Nederland
alex.gerdes@ou.nl

Johan Jeuring
Department of Information and Computing Sciences,
Utrecht University
J.T.Jeuring@uu.nl

Bastiaan Heeren
School of Computer Science,
Open Universiteit Nederland
bastiaan.heeren@ou.nl

ABSTRACT

We introduce an interactive tutor that supports the stepwise development of simple functional programs. Using this tutor, students receive feedback about whether or not they are on the right track, can ask for a hint when they are stuck, and get suggestions about how to refactor their program. Our tutor generates this semantically rich feedback from model solutions, using advanced concepts from software technology. We show how a teacher can add an exercise to the tutor, and fine-tune feedback. We report on an experiment in which we used our tutor.

Categories and Subject Descriptors

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

General Terms

Languages, Human Factors, Measurement

Keywords

Functional programming, Haskell, tutoring

1. INTRODUCTION

Introductory functional programming courses often start with distinguishing the various steps a student has to take to write a program. A teacher usually explains by example how to develop a program: give the main function a name; if the function takes an argument, give the argument a name; if the value of the argument determines the action to be taken subsequently, analyse the argument, and depending on the form of the argument, develop the appropriate right-hand sides, etc. Once a student starts developing a program herself, in a lab session or at home, this kind of explanatory help is usually not present. Moreover, giving immediate help to large classes of students is almost always impossible. Especially beginning programmers are often at a loss about how to proceed when developing a program. We introduce an interactive tutor that supports the stepwise development of simple functional programs in the lazy, pure, higher-order functional programming language Haskell [19]. Using this tutor, students learning functional programming develop their programs incrementally, receive feedback about whether or not they are on the right track, can ask for a hint when they are stuck, and get suggestions about how to refactor their program. The interactive tutor gives hints at each step, generates worked-out solutions for exercises, and recognises common errors made by students. All of this functionality is calculated automatically from the teacher-specified annotated solutions and non-solutions for a problem.

To support learning programming, many intelligent tutoring programs have been developed. There exist intelligent tutors for Prolog [11], Lisp [1], Pascal [13], Java [23], Haskell [15], and many more programming languages. Evaluation studies have indicated that

- 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 [5],
- using intelligent tutors requires less help from a teacher while showing the same performance on tests [18],
- using such tutors increases the self-confidence of female students [14],
- the immediate feedback given by many of the tutors is to be preferred over the delayed feedback common in classroom settings [17].

Despite the evidence for positive effects of using intelligent programming tutors, they are not widely used. An important reason is that building an intelligent tutor for a programming language is difficult and a substantial amount of work [20]. Some of these tutors are well-developed and extensively tested in classrooms, but most haven't outgrown the research prototype phase, and are not maintained anymore. Furthermore, deploying an intelligent tutor in a course is often hard for a teacher [2]. Most teachers want to adapt or extend an intelligent programming tutor to their needs. Adding an exercise to a tutor requires investigating which strategies can be used to solve the exercise, what the possible solutions are, and how the tutor should react to behaviour that doesn't follow the desired path. All this knowledge then has to be translated into the internals of the tutor, which implies a substantial amount of work. Other tutors have a fixed set of exercises, or enforce a strict order in which a program is constructed.

Our tutor is built on top of the Helium compiler for Haskell [10], which gives excellent syntax-error and type-error messages, and reports dependency analysis problems in a clear way. The most interesting feature of our tutor is that the hints and feedback given at intermediate steps are derived automatically from teacher-specified annotated solutions and non-solutions for a problem. This reduces the work required for using the tutor, and allows a teacher to use her favourite exercises. Furthermore, the order in which a student constructs a program using our tutor is quite flexible. The tutor is offered as a web application[1], which further reduces the burden to use it.

This paper describes a tutor that supports the step-wise, flexible development of simple functional programs, giving feedback and hints at intermediate steps, and showing worked-out examples. We make the following contributions:

- The feedback and hints are calculated automatically from teacher-specified annotated solutions for a problem.
- It discusses results of an experiment in which we used the tutor in a class-room setting.

This paper is organised as follows. Section 2 introduces our interactive functional programming tutor by means of an example session. Section 3 shows what a teacher has to do to add a programming exercises to the tutor. Section 4 discusses the result of using our tutor with a class of beginning functional programming students. Related and future work is discussed in Section 5, and Section 6 concludes.

2. AN INTERACTIVE SESSION

This section introduces our intelligent functional programming tutor by means of some interactions of a hypothetical student with the tutor. We assume that the student has visited lectures on how to write simple functional programs on lists.

The teacher has set a couple of exercises from the Ninety-nine Haskell Problems[2], in particular problem 22: Create a list containing all integers within a given range. We now show a couple of possible scenarios in which a student interacts with the tutor to solve this problem. At the start of a tutoring session the tutor gives a problem description:

```
Write a function that creates a list with all
integers between a given range:
```

$range :: Int \to Int \to [Int]$

```
For example:
```

$> range\ 4\ 9$
$[4, 5, 6, 7, 8, 9]$

and displays the name of the function to be defined, along with its parameters:

$range\ x\ y = \bullet$

[1]http://ideas.cs.uu.nl/ProgTutor/

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

The task of a student is to refine the holes, denoted by •, of the program. A student can use such holes to defer the refinement of parts of the program. After each refinement, a student can ask the tutor whether or not the refinement is bringing him or her closer to a correct solution, or, if the student doesn't know how to proceed, ask the tutor for a hint. Besides holes, 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 would respond with:

```
You can proceed in several ways:
- Implement range using the unfoldr function.
- Use the enumeration function from the
prelude[3].
- Use the prelude functions take and iterate.
```

We assume a student has some means to obtain information about functions and concepts that are mentioned in the feedback given by the tutor. This information might be obtained via lectures, an assistant, lecture notes, or even via the tutor at some later stage. The tutor can make a choice between the different possibilities, so if the student doesn't want to choose, and just wants a single hint, she gets:

```
Implement range using the unfoldr function.
```

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

$$\begin{array}{l} unfoldr \quad :: (b \to Maybe\ (a, b)) \to b \to [a] \\ unfoldr\ f\ b = \textbf{case}\ f\ b\ \textbf{of} \\ \qquad\qquad Just\ (a, b') \to a : unfoldr\ f\ b' \\ \qquad\qquad Nothing \quad \to [\,] \end{array}$$

The higher-order function *unfoldr* builds a list from a seed value, the second argument b. The first argument f is a producer function that takes the seed element and returns *Nothing* if it is done producing the list, or returns *Just* (a, b'), in which case, a is prepended to the output list and b' is used as the argument in the recursive call.

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

```
Define function range in terms of unfoldr,
which takes two arguments: a seed value, and
a function that produces a new value.
```

with the final bottom-out hint:

```
Define: range x y = unfoldr • •
```

At this point, the student can refine the function at two positions. In this exercise we do not impose an order on the sequence of refinements. However, the tutor offers a teacher the possibility to enforce a particular order of refinements. Suppose that the student chooses to first implement the producer function:

$$\begin{array}{l} range\ x\ y = unfoldr\ f\ \bullet \\ \quad \textbf{where}\ f\ i \mid \bullet = \bullet \end{array}$$

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

Note that the student has started to define the producer function in a **where** clause. She continues with the introduction of the stop criterion:

```
range x y = unfoldr f •
  where f i | i == y + 1 = •
```

There are several ways in Haskell to implement a condition. Here the student has chosen to define the function f with a so-called guarded expression; the predicate after the vertical bar acts as a guard. The student continues with:

```
range x y = unfoldr f •
  where f i | i == y + 1 = Just •
```

The tutor responds with:

```
Unexpected right hand side of f on line 3
```

Here the tutor indicates that the partial definition of f does not match any of the model solutions. Correcting the error, the student enters:

```
range x y = unfoldr f •
  where f i | i == y + 1 = Nothing
```

which is accepted by the tutor. If the student now asks for a hint, the tutor responds with:

```
Introduce a guarded expression that gives the
output value and the value for the next
iteration.
```

She continues with

```
range x y = unfoldr f •
  where f i | i == y + 1 = Nothing
            | otherwise = Just •
```

which is accepted, and then

```
range x y = unfoldr f •
  where f i | i == y + 1 = Nothing
            | otherwise = Just (n, i + 1)
```

which gives:

```
Error:  undefined variable n
```

This is an error message generated by the compiler. Our tutor displays the syntax and type errors messages generated by Helium. The student continues with:

```
range x y = unfoldr f x
  where f i | i == y + 1 = Nothing
            | otherwise = Just (i, i + 1)
```

which completes the exercise:

```
You have correctly solved the exercise.
```

A student can develop a program in any order, as long as all variables are bound. For example, a student can write

```
range x y = •
  where f i | • = •
```

and then proceed with defining f. This way, bottom-up developing a program is supported to some extent.

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 where the error appears to be,
- show a complete worked-out example.

The next section shows what a teacher has to do to achieve the functionality of the tutor as described in this section.

3. SPECIFYING EXERCISES

The interactions of the tutor are based on *model solutions* to programming problems. A model solution is a program that an expert writes, using good programming practices. A teacher adds a programming exercise to the tutor by specifying such model solutions. For our running example of calculating a range, we have specified three model solutions. The first model solution uses the enumeration notation from Haskell's prelude:

```
range x y = [x .. y]
```

The second model solution uses the prelude functions *take* and *iterate*:

```
range x y = take (y − x + 1) (iterate (+1) x)
```

The prelude function *iterate* returns an infinite list in which the next element is calculated by applying a given function, in this case a function that increases its argument by one, to the previous element, starting with a given value (x). The function *take* n returns the first n elements of a list. The last model solution uses the higher-order function *unfoldr* introduced in Section 2:

```
range x y = unfoldr f x
  where f i | i == y + 1 = Nothing
            | otherwise = Just (i, i + 1)
```

The tutor uses these model solutions to generate feedback. We not only recognise the exact specified model solution, but many variants. For example, although it appears entirely different, the following solution:

```
range x y =
  let f = λa → if a == y + 1
               then Nothing
               else Just (a, a + 1)
      g = λf x → case f x of
                   Just (r, b) → r : g f b
                   Nothing     → []
  in g f x
```

is recognised from the third model solution. To achieve this, we not only recognise the usage of a prelude function, such as *unfoldr*, but also its definition.

We use techniques and concepts from software technology, such as parsing, rewriting, and program transformations, to calculate semantically rich feedback [9]. In a nutshell, our approach is as follows [12]: we derive a *programming strategy* from the set of model solutions. This programming strategy is used to generate many variants of the model solutions. Before we compare a student solution to the solutions from the strategy, we normalise all solutions using program transformations. In this normalisation procedure we, for example, rewrite a **where**-clause into a **let**-expression.

3.1 Adapting feedback

It is important that a teacher can easily adapt the feedback given to a student. Our tutor offers the possibility to fine-tune the generated feedback by means of *annotating* model solutions. The remainder of this section shows a number of such annotations, accompanied with an explanation.

A description of a particular model solution can be added to the source code using the following construction:

$\{-\# \ \mathit{DESC}\ \ \mathit{Implement\ range\ using\ the\ unfoldr...}\ \ \#-\}$

The first hint in Section 2 gives the descriptions for the three model solutions for the range exercise. Next to a description for a single model solution, we can also give a description of the entire exercise. This description is given together with the model solutions in a configuration file for the exercise.

Another way to adapt the feedback is by specifying an *alternative* implementation for a prelude function. For example, the specification below shows how to give an alternative implementation for the *iterate* prelude function:

$$\begin{array}{l}\{-\# \ \mathit{ALT} \\ \quad \mathit{iterate}\ f = \mathit{unfoldr}\ (\lambda x \rightarrow \mathit{Just}\ (x, f\ x))\ \ \#-\}\end{array}$$

Using this annotation we not only recognise the prelude definition ($\mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x)$), but also the alternative implementation given here. By adding an alternative a teacher expands the number of accepted solutions and therefore changes the way in which the tutor gives feedback. Alternatives give the teacher partial control over which program variants are allowed.

Besides adding alternatives to expand the number of accepted solutions, a teacher may want to emphasise one particular implementation method. For example, a teacher may want to enforce the use of higher-order functions and prohibit their explicit recursive definitions. The *MUSTUSE* construction allows a teacher to disable the recognition of the definition of a prelude function:

$\mathit{range}\ x\ y = \{-\# \ \ \mathit{MUSTUSE}\ \ \#-\}\ \mathit{unfoldr}\ f\ x$

Another way to modify the response of the tutor is to add specific feedback messages at particular locations in the source code. For example:

$$\begin{array}{l}\mathit{range}\ x\ y = \\ \quad \{-\# \ \ \mathit{FEEDBACK}\ \ \mathit{Note...}\ \ \#-\}\ \mathit{take}\ (y - x + 1)\ \$ \\ \qquad \mathit{iterate}\ (+1)\ x\end{array}$$

Thus we give a detailed description of the *take* function. 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.

Adding an exercise to our tutor is relatively easy. To support managing exercises, they can be arranged in *classes*. Using a class a teacher groups together exercises, for example for practicing list problems, collecting exercises of the same difficulty, or exercises from a particular textbook.

4. EXPERIMENTAL RESULTS

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 at the lab session in which the students used the tutor. Table 1 shows the questions and the average of the answers on a Likert scale from 1 to 5. The first seven questions are related and indicate how satisfied a student is with the tutor. The last question addresses how students value the difficulty of the offered exercises.

The goal of the experiment is to analyse if students appreciate our approach, such as giving feedback on intermediate answers. The experiment does not check whether or not the tutor is more effective or efficient from a learning point of view. We hope to study this in the future.

Reflection on the scores.

The scoring shows 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 noticed this already 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 maybe sufficient, but there clearly is room for improvement.

4.1 Evaluation

In addition to questions about the usage of the tutor, the questionnaire contained a number of general questions, such as

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 current 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 this might not be true. It could well be that the program is incorrect or contains imperfections, such as being inefficient, and hence is rejected by our tutor. This remark addresses the fact that we cannot give feedback on a student program that deviates 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. First, the student program is incorrect. We should be able to detect this and give a counterexample. At the moment our tutor cannot do this, but we are working onincorporating testing, based on the QuickCheck [3] library. Second, the student program is correct and uses desirable programming techniques, but

#	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.

our tutor rejects it. In this case the set of model solutions should be extended with this solution. Third, the student program is functionally correct but contains some imperfections, such as, for example, a clumsy way of calculating the length of a list xs: $\mathit{length}\ (x : xs) - 1$. The tutor cannot conclude that a student program contains imperfections when it passes the tests but deviates from the strategy, so 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. Therefore, in this particular case we can give feedback that a student program probably has some undesired properties. We have used our approach for assessment of functional programming exercises [7], in which we could recognise almost 90% of the correct solutions based on only five model solutions. All of the other 10% of the correct solutions had some imperfections.

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 already addressed this problem and in the current version of the tutor it is not an issue anymore. We solved this problem by introducing a special search mode when recognising large steps.

In addition to the above experiment, we also asked a number of functional programming experts from the IFIP WG 2.1 group[4] 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. This suggestion corresponds to our own interpretation of the results from the experiment, and will be addressed in future work.

5. RELATED AND FUTURE WORK

There is a wealth of related work on intelligent programming tutors. We have not found other tutors that support the step-wise development of programs, automatically calculating feedback based on teacher-specified annotated solutions and non-solutions.

If ever the computer science education research field [6] finds an answer to the question of what makes programming hard, and how programming environments can support learning how to program, it is likely to depend on the age, interests, major subject, motivation, and background knowledge of a student. Programming environments for novices come in many variants, and for many programming languages or paradigms [8]. Programming environments like Scratch [21] and Alice [4] target younger students than we do, and emphasise the importance of constructing software with a strong visual component, with which students can develop software to which they can relate. We target beginning computer science students, who expect to work with real-life programming languages.

The Lisp tutor [1] is an intelligent tutoring system that supports the incremental construction of Lisp programs. At any point in the development a student can only take a single next step, which makes the interaction style of the tutor a bit restrictive. Furthermore, adding new material to the tutor is still quite some work. Using our approach, the interaction style becomes flexible, and adding exercises becomes relatively easy. Soloway [22] describes programming plans for constructing Lisp programs. These plans are instances of the higher-order function *foldr* and its companions. Our work structures the plans described by Soloway.

In tutoring systems for Prolog, a number of strategies for Prolog programming have been developed [11]. Strategies are matched against complete student solutions, and feedback is given after solving the exercise. We expect that these strategies can be translated to our situation, and can be reused for a programming language like Haskell.

Our work resembles the top-down Pascal editors developed in the Genie project [16]. These series of editors provide structure editing support, so that a student does not have to remember the particular syntax of a programming language. In our case students do have to write programs using the syntax of Haskell, but the intermediate steps are comparable. The Genie editors did not offer strategical support.

Our functional programming tutor grew out of a program assessment tool, which automatically assesses student programs based on model solutions [7] and program transformations to rewrite programs to normal form. Similar transformations have been developed for C++-like languages [24].

In the next couple of months we will add testing capabilities to our tutor. In addition to testing the student solution

[4]`http://www.cs.uu.nl/wiki/bin/view/IFIP21/WebHome`

against a model solution, we plan to offer the possibility to specify *properties*. A property is a statement that should always hold, and allows us to validate model solutions. For our running example, the following property states that the size of the generated list by *range x y* is equal to $y - x + 1$:

$$
\begin{array}{l}
\{-\#\ PROP \\
\quad prop\_range\ x\ y = \\
\qquad x \leqslant y \Longrightarrow length\ (range\ x\ y) == y - x + 1\ \ \#-\}
\end{array}
$$

Furthermore, we will extend the set of supported exercises. Once the tutor is sufficiently mature, we will add a service for teachers to upload their own annotated solutions and non-solutions.

6. CONCLUSIONS

We have introduced an intelligent tutor that supports the step-wise development of simple functional programs. A student can develop a program in many different ways. Our tutor automatically calculates hints and feedback at intermediate development steps from teacher-specified annotated solutions and non-solutions for a problem. This reduces the work required for using the tutor, and allows a teacher to use her favourite exercises.

We have conducted an experiment in which around a hundred students worked with our functional programming tutor. Furthermore, we asked functional programming experts about the design choices we made. The main conclusions of these two investigations are:

– Students appreciate worked-out solutions, and are moderately positive about the tutor.

– We need to judge student programs even when a student deviates from the model solutions. Therefore, we need to extend our tutor with testing capabilities.

Acknowledgements.

We thank Andres Löh and Doaitse Swierstra for allowing us to perform an experiment with our tutor in their classes. Furthermore, we thank Bram Vaessen for analysing the scores of our questionnaire.

References

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

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

[3] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP 2000: International Conference on Functional Programming*, 2000.

[4] M. J. Conway. *Alice: Easy-To-Learn 3D Scripting For Novices.* PhD thesis, University of Virginia, 1997.

[5] A. T. Corbett, J. R. Anderson, and E. J. Patterson. Problem compilation and tutoring flexibility in the Lisp tutor. In *Proceedings ITS'88: Intelligent Tutoring Systems*, pages 423–429, 1988.

[6] S. Fincher and M. Petre, editors. *Computer Science Education Research.* Routledge Falmer, 2004.

[7] A. Gerdes, J. Jeuring, and B. Heeren. Using strategies for assessment of programming exercises. In *SIGCSE*, pages 441–445, 2010.

[8] M. Guzdial. Programming environments for novices. In S. Fincher and M. Petre, editors, *Computer Science Education Research.* Routledge Falmer, 2004.

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

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

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

[12] J. Jeuring, A. Gerdes, and B. Heeren. A programming tutor for Haskell. In *Proceedings of CEFP 2011: Lecture Notes of the Central European School on Functional Programming*, LNCS. Springer, 2011.

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

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

[15] N. López, M. Núñez, I. Rodríguez, and F. Rubio. WHAT: Web-based Haskell adaptive tutor. In *AIMSA 2002: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 71–80. Springer-Verlag, 2002.

[16] P. Miller, J. Pane, G. Meter, and S. Vorthmann. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments*, 4(2):140–158, 1994.

[17] E. Mory. Feedback research revisited. In D. Jonassen, editor, *Handbook of research for educational communications and technology*, 2003.

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

[19] S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report.* Cambridge University Press, 2003. A special issue of the Journal of Functional Programming.

[20] N. Pillay. Developing intelligent programming tutors for novice programmers. *SIGCSE Bull.*, 35(2):78–82, 2003.

[21] M. Resnick et al. Scratch: programming for all. *Commun. ACM*, 52:60–67, Nov. 2009.

[22] E. Soloway. 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, 1985.

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

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