Teachers and students in charge — Using annotated model solutions in a functional programming tutor

Alex Gerdes

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2012-007

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

ISSN: 0924-3275

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

Teachers and students in charge

Using annotated model solutions in a functional programming tutor

Alex Gerdes[1], Bastiaan Heeren[1], and Johan Jeuring[1,2]

[1]School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
{lbyage, bhr, jje}@ou.nl
[2] Department of Information and Computing Sciences, Universiteit Utrecht

Abstract We are developing Ask-Elle, a programming tutor that supports students practising functional programming exercises in Haskell. Ask-Elle supports the stepwise construction of a program, can give hints and worked-out solutions at any time, and can check whether or not a student is developing a program similar to one of the model solutions for a problem. An important goal of Ask-Elle is to allow as much flexibility as possible for both teachers and students. A teacher can specify her own exercises by giving a set of model solutions for a problem. Based on these model solutions our tutor generates feedback. A teacher can adapt feedback by annotating model solutions. A student may use her own names for functions and variables, and may use different, but equivalent, language constructs. This paper shows how we can use annotated model solutions from a teacher to give feedback to a student in Ask-Elle. This requires both translating annotated model solutions to a form which we can use to track intermediate student steps, and developing techniques to avoid the state space explosion we get when analysing intermediate, incomplete, student answers.

Keywords: Functional programming, tutoring, Haskell

1 Introduction

Learning to program is challenging. The results of a first course in programming are often disappointing [21]. Learning by doing through developing programs, and learning through feedback [10] on these programs are essential aspects of learning programming. Students often get feedback late or not at all in a first programming course at university level, because students work outside class hours, or at home, and because these courses often attract quite a lot of students, where individual feedback in class is difficult or impossible, and corrected labs are returned to students weeks after they have been handed in.

To support learning programming, many intelligent programming tutors have been developed. Intelligent programming tutors support the development of programs, and can give immediate feedback to the student. There exist programming tutors for Lisp [3], Prolog [15], Java [17], Haskell [19], and many more

programming languages. Some of these tutors are well-developed tutors extensively tested in classrooms, others haven't outgrown the research prototype phase yet, and quite a few have been abandoned.

Evaluation studies have indicated that working with an intelligent tutor supporting the construction of programs may have positive effects. For example, using the LISP tutor is more effective when learning how to program than doing the same exercise "on your own" using only a compiler [7], and using a problem-solving software tutor increases the self-confidence of female students [18].

The functionality and help offered by programming tutors varies widely. The following aspects play a role:

- *Development process*: does the tutor support the incremental development of programs, where a student can obtain feedback or hints on incomplete programs, can a student follow his or her preferred way to solve a particular programming problem, does the tutor support refactoring a program, can a student submit a complete solution to a problem in the tutor?
- *Correctness*: does the tutor guarantee that a student solution is correct, can it check that a student has followed good programming practices, can it verify that the student solution has the desired efficiency, does it give an explanation why a program is incorrect, does it give counterexamples for incorrect programs, and/or does it detect at which point of a program a particular property is violated?
- *Adaptivity*: can a teacher add his or her own exercises to a tutor, and can (s)he adapt the behaviour so that particular ways for solving an exercise are enforced or disallowed?

None of the existing programming tutors addresses all of the above aspects. In particular, for tutors that support the incremental development of programs, it is usually quite hard for teachers to adapt or add programming exercises to the tutor, and to adapt the feedback given by the tutor.

Programming tutors are not used much. Anderson et al. [4] mention the lack of adaptability as one of the main reasons for the slow uptake of their tutors outside their own teaching environment. It is usually quite hard for teachers to adapt or add programming exercises to a programming tutor, and to adapt the feedback given by the tutor. Bokhove and Drijvers [6] list teacher adaptability as one of the four fundamental requirements for mathematical learning environments, and Lowes [20] found that among a group of almost a hundred teachers of on-line courses, almost 70% regularly adapted their assignments. Teacher adaptability is of fundamental importance for the uptake of learning environments.

Another important aspect of a programming tutor is that it offers sufficient freedom to students: a student should be able to use her own names, to use her own favourite programming style, her own refinement step-size, etc. Similar to the Lisp tutor [7], the refinement rules in our tutor model Haskell at the finest grain size that has functional meaning in Haskell, but we want to offer students the possibility to make larger steps than these small steps. This is challenging in the context of teacher annotated model solutions to program exercises.

This paper investigates how we can develop a programming tutor:

- 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, following the formative feedback guidelines to enhance learning [22], based on teacher-specified annotations in model solutions,
- 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.

In particular, we address some of the technical challenges that need to be solved to develop such a tutor.

This paper is organised as follows. Section 2 introduces our programming tutor, and shows how a teacher specifies a programming problem for the tutor, which a student solves in the tutor. Section 3 discusses how we can combine teacher-annotated model solutions to both give hints to students as well as diagnose partial student programs. Section 4 shows how we recognise student steps where step size doesn't matter. Section 5 discusses related work and concludes.

2 The Ask-Elle programming tutor

We are developing Ask-Elle, a programming tutor for Haskell [16]. See Figure 1 for a screenshot. Haskell is a lazy, higher-order functional programming language with a relatively simple semantics, and a limited core based on the λ-calculus. It is taught at many universities all over the world. More than 50.000 copies of introductory Haskell text books have been sold. Our tutor targets first year computer science students and offers introductory functional programming exercises. The tutor produces different types of feedback. It can:

- give hints, in increasing level of detail, when a student is stuck,
- analyse intermediate answers, and report whether or not a student is still on the right track,
- show how a complete program is constructed step by step,
- report type and syntax errors.

Suppose a teacher wants to set an exercise to develop a program that reverses a list. For example:

Data.List⟩ *reverse* `"A man, a plan, a canal, panama!"`
`"!amanap ,lanac a ,nalp a ,nam A"`
Data.List⟩ *reverse* $[1, 2, 3, 4]$
$[4, 3, 2, 1]$

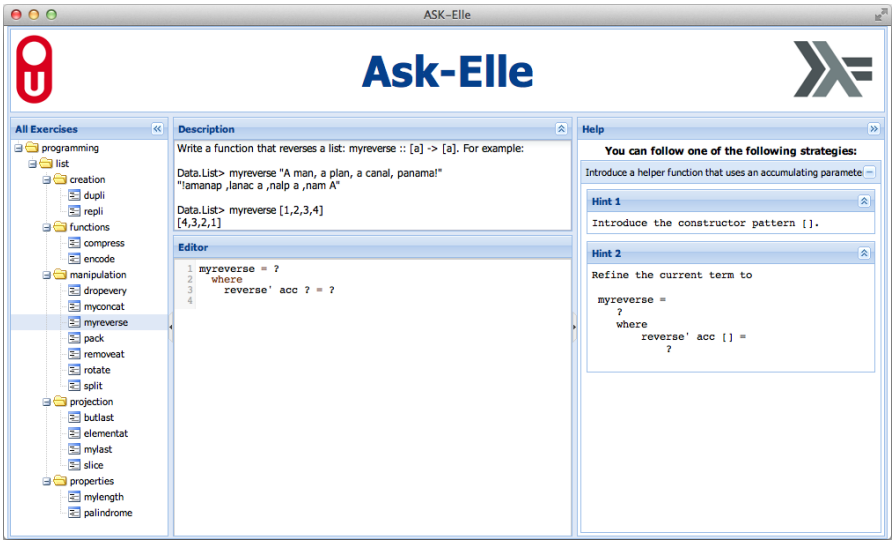


Figure 1. ASK-ELLE: a web-based functional programming tutor

The teacher specifies the exercise by means of three solutions:

{-# DESC Use the prelude function *foldl*. #-}
$reverse_1 = foldl$ {-# FEEDBACK Use *flip* and (:). #-}(*flip* (:)) []

{-# DESC Use explicit recursion. #-}
$reverse_2\ [] \quad = []$
$reverse_2\ (x : xs) = reverse_2\ xs +\!\!+ [x]$

{-# DESC Define a helper function with an accumulating parameter. #-}
$reverse_3\ list = reverse'\ list\ []$
where
$reverse'\ [] \quad reversed = reversed$
$reverse'\ (x : xs)\ reversed = reverse'\ xs\ (x : reversed)$

Note that the second solution is suboptimal, since it takes quadratic time in its input. Despite the fact that we can also specify suboptimal solutions, we will call these solutions "model solutions". A teacher may annotate the solutions to fine-tune the generated feedback. The three solutions above are annotated with a high-level description of the approach used in the solution, using {-# DESC .... #-}. In addition to the description annotation, the teacher has attached a specific feedback message to the operator argument of *foldl* in the first model solution, using {-# FEEDBACK ... #-}. When introducing the operator is one of the steps a student can take, and the student asks for help, the tutor will display the specified message. Since {-... -} is used for multi-line comments

in Haskell, annotated solutions are valid Haskell programs. Other classes of annotations are {-# ALT ... #-} and {-# MUSTUSE #-}. The ALT annotation specifies an alternative definition of a function, such as defining *map f xs* by means of a list comprehension $[f\ x \mid x \leftarrow xs]$. The tutor also accepts the definition of a library function in addition to its usage; the MUSTUSE annotation disallows this behaviour. A feedback annotation binds stronger than all other language constructs, i.e., it applies to the smallest possible expression. In the following example:

$$reverse_1 = \{\text{-\# MUSTUSE \#-}\}foldl\ (flip\ (:))\ [\,]$$

the student is not allowed to use the explicit recursive definition of *foldl* due to the MUSTUSE annotation. The student is allowed to use the definition of *flip*, because the scope of the MUSTUSE annotation is limited to the expression *foldl*. If a teacher wants to prohibit the use of definitions of library functions altogether, she can expand the scope of the annotation by placing parentheses:

$$reverse_1 = \{\text{-\# MUSTUSE \#-}\}(foldl\ (flip\ (:))\ [\,])$$

Suppose a student chooses to implement the *reverse* function using the higher-order function *foldl*, and has started in the following way:

$$reverse = foldl\ \bullet\ \bullet$$

A hole (•) denotes an incomplete part of a program that needs to be refined. Here the student can refine the program at two locations: she can introduce an operator argument, or a base argument. If the student asks for help, the tutor responds with:

`Use` *flip* `and` (:)`.`

Since refining the operator argument is one of the possible next steps, the hint specified in the annotated solution is given. If the student asks for a more detailed hint she gets:

`Introduce the function` *flip*`.`

If the student remains stuck, we can give a bottom-out hint:

`Refine the program to:`

$$reverse = foldl\ (flip\ \bullet\ )\ \bullet$$

In addition to giving detailed feedback about a single step, the tutor can list the different steps that are allowed. In the given example, refining the hole for the base case is another step that the student can take. When asked for a hint about another step the tutor responds with:

`Introduce the empty list constructor` [`]`.

3 The teacher in charge

Our tutor takes a set of model solutions for a programming exercise as input. Using these solutions, it constructs a programming strategy [8], which it uses to follow a student when incrementally solving the programming exercise. The strategy is interpreted as a recogniser that recognises program refinement steps of students. This section discusses how we construct a recogniser from several possible model solutions, such that teacher annotations in model solutions are used when giving feedback, hints, or worked-out solutions.

3.1 Strategy language

Heeren et al. [12] have developed a strategy language for describing procedural skills as rewrite strategies. The basic elements of the strategy language are rewrite and refinement rules, and the language supports combining strategies by means of *strategy combinators*. We use the following strategy combinators:

r	symbol: a rewrite or refinement rule
$s \mathbin{<|>} t$	choice: do either s or t
$s \mathbin{<\star>} t$	sequence: do s before t
fail	always fails (the unit of choice)
succeed	always succeeds (the unit of sequence)
label ℓ s	attach a label ℓ to a strategy (to mark a position)
fix f	fixed-point operator for recursion
$s \mathbin{<\%>} t$	interleave: steps from s and t are performed in any order
$s \mathbin{\%>} t$	left-interleave: start interleaving with a step from s
$\langle s \rangle$	atomic: s cannot be interleaved

where s and t are strategies and f is a function that takes a strategy as argument, and returns a strategy. The *label* combinator takes a label and a strategy as arguments, and offers the possibility to attach a text to the argument strategy. Using these combinators we can express the order of refinements that are necessary to define a program. For example, the strategies for the different model solutions are combined into a single strategy using the choice combinator.

We convert a model solution to a programming strategy using a tree matching algorithm to translate every construct in the abstract syntax tree of a model solution to a corresponding refinement rule. For example, we have a constructor for variables, and hence also a refinement rule for refining a hole to a variable. A feedback annotation in the model solution leads to a label in the derived strategy, an alternative annotation adds a choice with the strategy derived for the alternative, and a MUSTUSE annotation removes a choice. For instance, the model solution $reverse_1$ is translated to the following (simplified) strategy:

```
fun <*> app <*> (app <*> var "foldl"
                     <%> label "..." (app <*> var "flip"
                                          <%> var "(:)"))
                <%> var "[]"
```

where *fun*, *app*, and *var* are refinement rules for the introduction of a function, an application, and a variable respectively. The interleaving operator <%> allows refining the arguments of *foldl* in any order. The feedback annotation in $reverse_1$ has been translated to a label at the location of the annotation in the model solution. We distinguish two types of annotations: location specific and global annotations. Location specific annotations, such as the feedback message annotation, target a particular expression in the model solution. To obtain this location information, and the attached feedback, we have extended the Helium [13] compiler that we use for compiling the source code. The lexer, parser, and abstract syntax have been extended to incorporate feedback annotations. The global annotations are always placed in the header of the model solution source file. These annotations concern the entire model solution, such as the description annotation.

3.2 Strategy recogniser

We interpret a strategy as a context-free grammar. The language generated by a strategy can be used to determine whether or not a sequence of rules applied by a student follows a strategy. The sequence of rules should be a sentence in the language, or a prefix of a sentence, since we solve exercises incrementally. A recogniser for a context-free grammar recognises refinement steps that are applied to some initial term, usually the empty program. The recogniser maintains the current location within the strategy at which the student has applied a refinement rule, to give precise feedback. Using the information about the progress of a student, we can calculate which steps are allowed next, and check whether or not a student deviates from a path towards a model solution.

The recogniser maintains the active labels, which contain the texts that are used when a student asks for a hint. The interpretation of a strategy with a label introduces the special rules ENTER and EXIT, parameterised by the label. These rules are only used for tracing positions in strategies, and delivering feedback texts when necessary. A label is active when we have recognised the ENTER rule of that particular label, but not yet its corresponding EXIT rule.

Most of the feedback is derived from the grammar functions *empty* and *firsts*. The *empty* function determines whether or not the language described by a strategy contains the empty sentence. The *firsts* function determines the set of rules with which a sentence in the language of a strategy can start. These grammar functions are used to give hints to students.

3.3 Parallel top-down recogniser

The recogniser recognises prefixes, and hence also accepts intermediate (incomplete) solutions. It cannot use backtracking, since this would imply that it accepts steps that do not lead to a solution, and hence guides a student into the wrong direction. It follows that the recogniser needs to choose between the various model solutions on the basis of a single refinement step. This is problematic

when multiple model solutions share a first step, i.e., when we encounter a left-factor in the strategies generated for the model solutions. Note that combining model solutions almost always leads to left-factors. The introduction of a declaration, and a function name is very often shared between the different model solutions. Consider the following somewhat contrived strategy:

$$\begin{aligned} \mathit{leftFactor} \;=\; & \mathit{label}\ \ell_1\ (\mathit{app} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"f"} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"x"}) \\ & \mathbin{<|>}\ \mathit{label}\ \ell_2\ (\mathit{app} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"g"} \mathbin{<\!\star\!>} \mathit{var}\ \texttt{"y"}) \end{aligned}$$

The two sub-strategies labelled ℓ_1 and ℓ_2 share a left-factor: the refinement rule *app*. We should decide which sub-strategy to follow *after* recognising the application of *app*, but the requirement to choose based on a single refinement step does not allow for this. The standard method of dealing with this problem is to apply left-factoring, a grammar transformation that removes left-factors. However, the presence of labels makes it more difficult to use left-factoring, since moving or merging labels leads to scrambling annotations of model solutions, making it very hard if not impossible to give the intended hints. We need to defer committing to a particular path in the strategy.

To deal with left-factors, we fork the recogniser whenever we run into a left-factor. If any of these recognisers fails to recognise the student solution, we discard it. Thus we obtain a top-down variant of a parallel recogniser. Using a top-down parallel recogniser we allow a teacher to specify model solutions that have common components.

The strategy language has also been used to describe how to solve exercises in many mathematical domains, such as solving quadratic equations, and differentiating functions. The strategies in these domains do not contain left-factors, and a top-down recogniser for LL(1) grammars that are not left-recursive supports solving such exercises well.

4 The student in charge

Most model solutions are programs that an expert would write; they make use of good programming practices. During the stepwise definition of a solution, we guide a student towards one of these model solutions, based on the derived programming strategy. The derived programming strategy, however, is rather strict: it only accepts solutions that are syntactically equivalent to one of the model solutions. We want to also accept all kinds of variants of model solutions, and have taken a number of measures to increase the number of accepted solutions.

The first measure is the introduction of *standard strategies*. These standard strategies are defined for functions from standard libraries such as the Haskell prelude, and recognise the usage of a standard function as well as its definition. For example, the standard strategy for the *flip* function not only recognises *flip* (:), but also the equivalent $\lambda x\ y \to y : x$. If a teacher wants to enforce the usage of a standard function, and not its definition, she can use the {-# MUSTUSE #-} annotation. However, there are also syntactic differences

that we cannot or would not like to capture in a strategy. For example, we allow a student to use different variable names. To ignore such differences we use *program normalisation*. Our normalisation procedure uses program transformations to rewrite a program to a normal form. We use amongst others inlining, α-renaming, β- and η-reduction, and desugaring program transformations. Our normalisation performs β- and η-reductions in applicative order and normalises a program to β-normal form. The procedure we use is related to normalisation by evaluation [5].

We have performed several experiments with Ask-Elle and asked students to evaluate the programming tutor [9]. We conducted the experiments in a course on functional programming for second year bachelor students at Utrecht University in September 2011. The course attracted more than 200 students. Around a hundred students used our tutor in two sessions, and 40 of them filled out a questionnaire about the tutor. The goal of the experiment was to find out if students appreciate our approach, such as giving feedback on intermediate answers. We did not investigate whether or not the tutor is more effective or efficient from a learning point of view. We hope to study this in the future.

Students were generally positive about using the tutor; their main comment was that the tutor is of no help when performing many refinement steps in a single step. Some students even pasted complete solutions in the tutor, which we might consider undesirable behaviour, but which we don't want to disallow. At the time of the experiments, the tutor could only recognise a limited number of steps towards a solution when a student submitted a (possibly partial) program. The enhancements described in the remainder of this section lift this restriction.

Recognising multiple steps is difficult. In an expression such as

$$
\begin{array}{l}
\mathit{reverse}\ \mathit{list} = \mathit{reverse}'\ \mathit{list}\ [\,] \\
\quad \textbf{where}\ \mathit{reverse}'\ \ \bullet_a \qquad\quad \bullet_b \ \ = \bullet_c \\
\qquad\qquad \mathit{reverse}'\ (\ \bullet_d\ \ :\mathit{xs})\ \ \bullet_e \ \ = \mathit{reverse}'\ \mathit{xs}\ \ \bullet_f
\end{array}
$$

a student may refine any of the five holes, in any order. The derived strategy for this solution allows to interleave the refinement of the five holes. The number of sentences recognised by this strategy is enormous. For example, the result of interleaving a more restricted strategy *abc* that recognises the sequence of the refinement rules a, b, and c, with a strategy *def* (that is, *abc* <%> *def*) results in the following set:

$$
\begin{array}{l}
\{\, \mathit{abcdef}, \mathit{abdefc}, \mathit{abdecf}, \mathit{abdcef}, \mathit{adefbc}, \mathit{adebcf}, \mathit{adebfc}, \mathit{adbcef}, \mathit{adbefc}, \\
\ \ \mathit{adbecf}, \mathit{defabc}, \mathit{deabcf}, \mathit{deabfc}, \mathit{deafbc}, \mathit{dabcef}, \mathit{dabefc}, \mathit{dabecf}, \mathit{daefbc}, \\
\ \ \mathit{daebcf}, \mathit{daebfc}\,\}
\end{array}
$$

The number of interleavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$, see [11] for more details. This number grows quickly with longer sentences. That means that even for relatively small introductory programs the number of intermediate solutions is huge. In the case that all possible steps can be interleaved, as is not uncommon in programming exercises, the number of interleavings becomes close to $n!$.

The use of standard strategies not only increases the number of accepted solutions, but also the number of possible interleavings. Our experiment showed that it is not an option to check, by means of multiple calls to the *firsts* function, if a student submission is an element of the set of all possible intermediate solutions.

4.1 Pruning

We constrain the search space of intermediate answers to determine whether or not a student submission follows a strategy. First, we observe that the first steps of the different strategies for model solutions may be the same, but they diverge after a number of steps. For example, if a student submits

$$\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ \bullet$$

we know she follows the strategy of the first model solution, and we ignore the intermediate answers of the other model solutions, i.e., $\mathit{reverse}_2$ and $\mathit{reverse}_3$. Since we use refinement rules, a student can no longer refine her program towards those model solutions. This reduces the number of interleavings significantly. We filter out these intermediate answers by determining whether or not the normalised abstract syntax trees of the model solution and the student submission overlap, where a hole ($\bullet$) overlaps with any tree. We use depth-first search to find matching solutions, since it is more likely that a student first finishes a particular part of the program, such as a case alternative, than doing refinements at arbitrary places.

4.2 A search mode for the interleave combinator

Although pruning is a step forward, it is not good enough. Even with pruning, the search space remains too large, due to the amount of possible interleavings. To reduce the number of interleavings, we observe that when recognising multiple steps, the *order* of refinements of holes that may be interleaved is irrelevant. Consider the example from the previous subsection. For recognition it does not matter whether we first introduce the cons operator (:) followed by the empty list constructor [ ], or vice versa. Interleaving causes many duplicates in the set of intermediate answers. Consider the following derivation for our running example:

$$\begin{aligned} & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ \bullet \\ \Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ \bullet \\ \Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,] \end{aligned}$$

We can reach the same result with a different order of steps:

$$\begin{aligned} & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ \bullet \\ \Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ \bullet\ )\ [\,] \\ \Rightarrow\ & \mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,] \end{aligned}$$

We use the irrelevance of refinement order when recognising multiple steps by introducing a *search mode* for the interleave combinator. The semantics of the original interleave combinator chooses between the left-interleave of both sub-strategies:

$$x \quad <\%> y = (x \ \%> y) <|> (y \ \%> x)$$
$$(\langle a \rangle <\star> x) \ \%> \ y = a <\star> (x <\%> y)$$

The search mode for interleave changes the semantics of $<\%>$, which we denote by $<\%>^\bullet$. The changed combinator chooses between left-interleaving the left sub-strategy with the right sub-strategy, or taking the (non-interleaved) right sub-strategy:

$$x \quad <\%>^\bullet y = (x \ \%>^\bullet y) <|> y$$
$$(\langle a \rangle <\star> x) \ \%>^\bullet \ y = a <\star> (x <\%>^\bullet y)$$

In the first line, the right-hand side of the choice does not recognise steps from x. We recognise intermediate answers containing steps from x with the left-interleave of x with y. Because of the left-interleave, steps from x are recognised before steps from y. This is safe because the order of refinement steps does not matter. Using the search mode for interleave, all sequences of refinement steps leading to the same intermediate program are replaced by a single sequence, drastically reducing the search space. Recall the intermediate program example for *reverse* at the beginning of this section. The right-hand side diagram in Figure 2 shows that in search mode every (intermediate) program can be constructed by exactly one sequence. All other sequences depicted in the left-hand side diagram do not appear anymore when using the search mode. For generating hints we still use the normal behaviour of the interleave combinator.

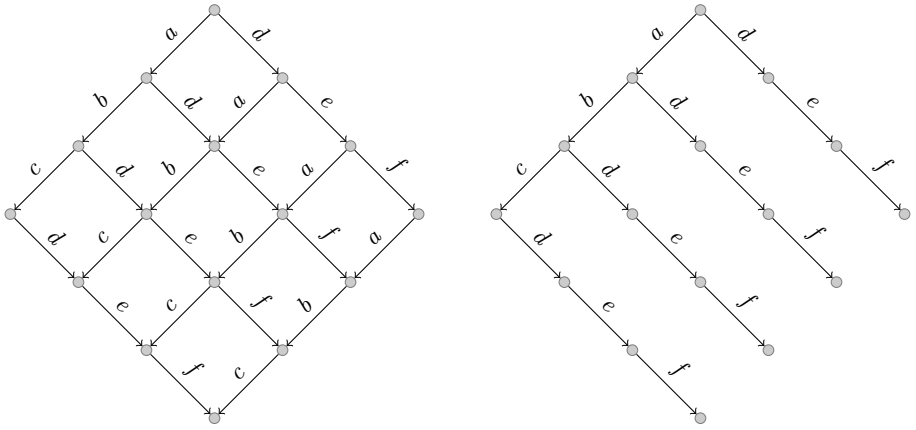
Our approach is similar to partial-order reduction in model checking [2]. It can be applied in the functional programming domain because we use refinement rules. If we would also use rewrite rules, we would need to prove that the rewriting system is Church-Rosser before we can use the alternative semantics of interleave.

5 Conclusions and related work

We have discussed two important issues for Ask-Elle, a programming tutor for Haskell.

First, we have shown how teachers can add programming exercises to our programming tutor by means of annotated model solutions. Teachers determine which solutions are accepted and/or suggested to students, and which solutions are not allowed. Since we do not want to give hints that do not lead to a solution, we cannot use backtracking or problem compilation [7] in our framework, and instead we introduce parallel top-down recognition.

Second, we have shown how we recognise almost arbitrary many student steps on the way to a solution. A student may take refinement steps in any order, but when recognising student steps we fix the order to reduce the search space.



(a) Normal mode (b) Search mode

Figure 2. Interleaving the sentences *abc* and *def*.

The concepts we have introduced to deal with these issues are not specific for Haskell. We can use the approach described in this paper to develop similar programming tutors for other functional programming languages, such as Lisp or OCaml. We believe that we have not made assumptions that exclude imperative programming languages, but we would have to further investigate this.

We have not yet performed experiments with teachers using our system, except for ourselves using the system. We want to perform experiments to test the new functionality of our tutor.

Our tutor resembles the Lisp tutor [3] in that it supports the stepwise development of programs, and gives hints at intermediate steps. By generating strategies from model solutions we think it is easier to add programming exercises to our tutor. Moreover, teachers can easily fine-tune the generated feedback.

The Prolog tutoring system [15] supports a number of strategies for Prolog programming. These strategies are matched against complete student solutions, and feedback is given after solving the exercise. Our tutor is able to give feedback during the stepwise construction of a program.

J-Latte [14] verifies complete student Java programs against constraints. In the future we want to add the possibility to check constraints on an incomplete student program to our tutor.

CTAT [1] is a framework for building intelligent tutoring systems based on example derivations. It mainly targets mathematical domains that are taught at high school level. The underlying technology is similar to our strategy language. Other similarities are that feedback messages can be attached to examples, and ambiguity is solved by maintaining multiple interpretations of student behaviour in parallel. A difference with our approach is that a procedure for solving an exercise is derived from an example derivation instead of a set of model solutions. CTAT supports derivations with rewrite rules, but not with refinement rules as

used in our programming domain. We have not found programming tutors built using CTAT.

References

1. V. Aleven, B. M. McLaren, and J. Sewall. Scaling up programming by demonstration for intelligent tutoring systems development: An open-access web site for middle school mathematics learning. *IEEE Transactions on Learning Technologies*, 2:64–78, 2009.
2. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18:97–116, 2001.
3. J. R. Anderson, F. G. Conrad, and A. T. Corbett. Skill acquisition and the LISP tutor. *Cognitive Science*, 13:467–505, 1986.
4. J. R. Anderson, A. T. Corbett, K. R. Koedinger, and Ray Pelletier. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4(2):167–207, 1995.
5. U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *LNCS*, pages 624–624. Springer Berlin / Heidelberg, 1998.
6. C. Bokhove and P. Drijvers. Digital Tools for Algebra Education: Criteria and Evaluation. *Int. Journal of Comp. for Math. Learning*, 15(1):45–62, April 2010.
7. A. T. Corbett, J. R. Anderson, and E. J. Patterson. Problem compilation and tutoring flexibility in the LISP tutor. In *Proceedings of ITS 1988: 4th International Conference on Intelligent Tutoring Systems*, pages 423–429, 1988.
8. A. Gerdes, B. Heeren, and J. Jeuring. Constructing Strategies for Programming. In José Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *Proceedings of the First International Conference on Computer Supported Education*, pages 65–72. INSTICC Press, March 2009.
9. A. Gerdes, J. Jeuring, and B. Heeren. An interactive functional programming tutor. In *Proceedings of ITICSE 2012: the 17th Annual Conference on Innovation and Technology in Computer Science Education*, 2012. To appear. Also available as technical report Utrecht University, UU-CS-2012-002.
10. J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.
11. B. Heeren and J. Jeuring. Interleaving strategies. In *Proceedings of MKM 2011: the 10th International Conference on Mathematical Knowledge Management*, volume 6824 of *LNAI*, pages 196–211. Springer, 2011.
12. B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010.
13. B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *Proceedings of Haskell 2003: the ACM SIGPLAN workshop on Haskell*, pages 62 – 71. ACM, 2003.
14. Jay Holland, Tanja Mitrovic, and Brent Martin. J-Latte: a constraint-based tutor for Java. In *Proceedings of ICCE 2009: the 17th International on Conference Computers in Education*, pages 142–146, 2009.
15. J. Hong. Guided programming and automated error analysis in an intelligent Prolog tutor. *Int. Journal on Human-Computer Studies*, 61(4):505–534, 2004.
16. 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, 2012. To appear.

17. M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4), 2003.
18. A. N. Kumar. 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, 2008.
19. N. López, M. Núñez, I. Rodríguez, and F. Rubio. WHAT: Web-based Haskell adaptive tutor. In *Proceedings of AIMSA 2002: the 10th Int. Conf. on Artificial Intelligence: Methodology, Systems, and Applications*, pages 71–80. Springer, 2002.
20. S. Lowes. Online teaching and classroom change: The impact of virtual high school on its teachers and their schools. Technical report, Columbia University, Institute for Learning Technologies, 2007.
21. M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR 2001, pages 125–180. ACM, 2001.
22. Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1):153–189, 2008.