Strategy Feedback in an E-learning Tool for Mathematical Exercises

Johan Jeuring

Wouter Pasman

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2007-007

www.cs.uu.nl

ISSN: 0924-3275

Strategy Feedback in an E-learning Tool for Mathematical Exercises

Johan Jeuring[1] and Wouter Pasman[2]

[1] ICS, Utrecht University, and Computer Science, Open University, the Netherlands,
johanj@cs.uu.nl
[2] EEMCS, Delft University of Technology, Delft, the Netherlands,
w.pasman@tudelft.nl

Abstract Exercises in mathematics are often solved using a standard procedure, such as for example solving a system of linear equations by subtracting equations from top to bottom, and then substituting variables from bottom to top. Students have to practice such procedural skills: they have to learn how to apply a particular strategy to an exercise. E-learning systems offer excellent possibilities for practicing procedural skills. The first explanations and motivation for a procedure that solves a particular kind of problems are probably best taught in a class room, or studied in a book, but the subsequent practice can often be done behind a computer. There exist many e-learning systems or intelligent tutoring systems that support practicing procedural skills. The tools vary widely in breadth, depth, user-interface, etc, but, unfortunately, almost all of them lack sophisticated techniques for providing immediate feedback. If feedback mechanisms are present, they are hard coded in the tools, often even with the exercises. This situation hampers the usage of e-learning systems for practicing mathematical skills. This paper introduces a formalism for specifying strategies for solving exercises. It shows how a strategy can be viewed as a language in which sentences consist of transformation steps. Furthermore, it discusses how we can use advanced techniques from computer science, such as term rewriting, strategies, error-correcting parsers, and parser combinators to provide feedback at each intermediate step from the start towards the solution of an exercise. Our goal is to obtain e-learning systems that give immediate and useful feedback.

1 Introduction

Many mathematical exercises are solved using a *strategy*. For example, to answer the question: 'What is the value of $(4 + 8 * 2)/5$' we apply rules for evaluating operators, until no such rule can be applied anymore.

$$\begin{aligned} & (4 + 8 * 2)/5 \\ \Rightarrow\ & (4 + 16)/5 \\ \Rightarrow\ & 20/5 \\ \Rightarrow\ & 4 \end{aligned}$$

Here $a \Rightarrow b$ means: a is rewritten into b. Other examples of strategies are:

- reducing a logical expression to disjunctive normal form by first pushing $\neg$'s over $\vee$'s and $\wedge$'s using de Morgan's rules, until they are in front of literals, and then distributing $\wedge$ over $\vee$, and
- solving a system of linear equations by subtracting equations from top to bottom, and then substituting variables from bottom to top.

For almost all mathematical exercises, at any educational level, students have to learn to apply a strategy to solve a particular class of exercises. Learning a strategy is sometimes also called practicing procedural skills, but since the term procedural skills is sometimes also used to refer to the capability to apply individual rewrite rules, we will use learning to apply a strategy instead of practicing procedural skills in the technical part of this paper.

E-learning systems offer excellent possibilities for practicing procedural skills. The first explanations and motivation for a procedure that solves a particular kind of problems are probably best taught in a class room, or studied in a book, but the subsequent practice can often be done behind a computer.

There exist many e-learning systems or intelligent tutoring systems that support practicing procedural skills. The tools vary widely in breadth, depth, user-interface, etc, but, unfortunately, almost all of them lack sophisticated techniques for providing immediate feedback. If feedback mechanisms are present, they are hard coded in the tools, often even with the exercises. This situation hampers the usage of e-learning systems for practicing procedural skills. This paper investigates techniques for providing flexible and immediate feedback in tools that support practicing procedural skills. The tools we envisage are interactive tools, in which a student gets an exercise, which consists of an expression (a structured object) from a certain domain. To solve the exercise, the student applies transformations to the expression, until a solution is reached. At each step, we want to be able to give feedback if the student does not follow the prescribed strategy correctly. For example, in the example of reducing a logical expression to disjunctive normal form, first all $\neg$'s not in front of a variable have to be eliminated before we want to remove all $\vee$'s below top level. If a student starts with removing $\vee$'s below top level, while there are still $\neg$'s to be eliminated, we want to tell the student that (s)he should first eliminate the $\neg$'s, before starting to remove $\vee$'s below top level.

We show how we can automatically construct feedback at each intermediate step from the following components:

- A domain description (for example logical expressions, or systems of linear equations). For a domain we need both the *abstract syntax* (what is the structure of expressions, for example `frac (int 1) (int 2)`), and the *concrete syntax* (how are the expressions visually presented to the student, for example $\frac{1}{2}$). We will assume a domain consists of trees.
- Rules for the domain (multiplication distributes over addition, zero is the unit of addition). This also includes basic evaluation rules, such as $3 + 5$ equals 8.

– 'Buggy rules' for the domain. Buggy rules represent common misconceptions (addition distributes over multiplication might be such a rule).
– A strategy for solving an exercise in the domain (subtract equations from top to bottom, substitute from bottom to top).

We can construct feedback by viewing each strategy as a *language*, where the alphabet consists of the transformation steps a student can apply. A *sentence* of a language for a particular strategy is a sequence of transformation steps, which transform a given problem into its solution. Our feedback engine checks that at each intermediate step, the sequence of transformation steps submitted until then by the student is a prefix of a sentence in the language of the strategy. Using techniques from the field of parser generators, parser combinators, and error-correcting parsers, we can automatically construct feedback if a step submitted by the student is not valid according to the strategy.

This paper has the following contributions:

– It claims that if e-learning tools are going to provide good feedback, it is necessary to explicitly specify the strategy for solving an exercise.
– It shows how any particular strategy can be viewed as the specification of a language, the sentences of which consist of sequences of transformation steps which turn a particular problem into its solution.
– It shows how we can give feedback to a student based on the strategy description and the transformation steps the student has taken.
– It introduces a formalism for specifying strategies for solving mathematical exercises. This formalism is similar to context-free grammars, but it contains constructs specific to strategies for solving exercises.

As far as we are aware, this approach to strategy specification for mathematical exercises is original. The approach may lead to considerably better feedback in e-learning tools.

This paper is organized as follows. Section 2 introduces the problem in more detail, and discusses related work. Section 3 shows an example domain with rules and a strategy in detail: namely logical expressions together with rules like de Morgan, a strategy for rewriting logical expressions to disjunctive normal form. Section 4 introduces a language for specifying strategies. Section 5 shows how we can construct feedback for a student given an exercise that should be solved according to a particular strategy. Section 6 concludes.

2 Feedback and e-learning

Procedural skills. When studying mathematics, students have to acquire, amongst others, procedural skills. Problems are often solved using a standard procedure, such as for example solving a system of linear equations by subtracting equations from top to bottom, and then substituting variables from bottom to top. Procedural skills appear at any educational level. For example, at the primary school level, pupils have to learn how to calculate the value of an expression such

as $3 * (4 + 5)$, or $\frac{6+7}{2+3} + \frac{8}{5-2}$; at secondary school level, students have to solve sets of linear equations; and at university level, students have to simplify logical expressions.

What are procedural skills? In this paper, we consider a procedural skill to be the ability to apply a number of manipulations to a structured object following a prescribed procedure. This includes the typical skills, such as manipulating mathematical objects, practiced at schools and universities. But not, for example, skills needed for flying [17]. A single manipulation of an object might be a rewrite step ('distribute multiplication over addition'), or it might be an evaluation step ('replace $a + b$ by its sum', which might also be considered as a rewrite step). A procedure describes how basic steps may be combined to solve a particular problem. A procedure is often called a *strategy* (or 'meta-level reasoning', 'meta-level inference' [8], 'procedural nets' [6], 'plans', 'tactics', etc.), and we will use this term in the rest of this paper. Strategies range from very simple, for example describing that a simple arithmetic expression with constants, + and − has to be simplified, to very complicated, describing a complicated procedure for solving an exercise from linear algebra.

Learning strategies. Strategies are usually taught in class. A teacher gives several examples of how to use a strategy on an example, and then lets students practice on examples. Students learn to apply a strategy both by the examples and explanation given by the teacher, as well as the individual practice on example problems. A student practices with exercises, makes errors, gets feedback, and possibly a renewed explanation, and uses the feedback to make progress. Feedback plays a very important rôle in learning [22,28]. It is physically impossible to give each student immediate feedback when practicing, so feedback on errors in applying a strategy is usually only given long after an error is made. It is quite common that a teacher provides feedback on worked out exercises on paper, often many days after the exercise has been solved.

E-learning systems and strategies. E-learning systems offer excellent possibilities for practicing with applying strategies. The first explanations and motivation for a strategy that solves a particular kind of problems are probably best taught in a class room, or studied in a book, but the subsequent practice can often be done behind a computer. The big advantage of using an e-learning system for practicing strategies is that such a system can provide immediate feedback [27]. Furthermore, it can use information it gathers about a student to select appropriate exercises for the student, it can analyze the behavior of the student to report possible sources of misconception, and it can inform a teacher about misconceptions that appear often in a group of students.

Existing e-learning systems for practicing strategies. Many e-learning systems or intelligent tutoring systems support practicing strategies. We have investigated two domains in some depth: tools for practicing different mathematical domains, such as calculus and algebra (many tools), and tools for teaching logic [11]. The

tools vary widely in breadth, depth, user-interface, etc, but, unfortunately, almost all of them lack sophisticated techniques for providing immediate feedback. Among the tools that provide feedback, we have encountered several classes.

First, there are many tools that only look at the final answer of a student. Most of these tools only return correct or incorrect. Some of these tools derive feedback from analyzing the answer [5,18,31,26]. For example, when the question is: 'Suppose you have 125 euros. You give 25 euros to a friend, and keep the remaining 100 euros. What percentage of the original 125 euros do you have left?' If the answer given is 75%, the feedback tries to say something about the possible misconception. Some multiple-choice tools specify the feedback per question with each answer to the question.

Tools like BUGGY [6], DEBUGGY [7], and BUGFIX [19] assume the existence of complete sets of correct and incorrect rewrite rules to diagnose bugs in students' solutions. These tools also look at the final answer of the student, but they try to derive the most probable incorrect intermediate rule that has been applied.

There are fewer e-learning tools that let a student solve an exercise in a stepwise fashion. Tools like Aplusix [9] and the Freudenthal applets [13] only report whether or not an intermediate rule has been applied correctly or not. Our own interactive exercise assistants for solving systems of linear equations [29], and rewriting logical expressions to disjunctive normal form [24], try to determine which rewrite rule has been applied by the student. If they cannot determine such a rule, they find the closest possible match, and give that as feedback. The feedback does not have to be specified per exercise, but is derived generically. ActiveMath [10,14] also gives feedback at each intermediate step, but the feedback is either incorrect/correct, or the feedback has to be specified per exercise. Finally, a tool like Math(X)Pert [4,3] only allows correct transformation steps, but can give hints for the next step to take.

All of these approaches have some problems. In the first approach the effort required to specify feedback is substantial: the size of the description of the exercise might easily increase with a factor ten. Furthermore, it is hard to reuse feedback across exercises, and if a teacher thinks of a better way to provide feedback, for example because (s)he has found a common misconception of the students, all exercises that use this feedback have to be adapted. The second approach is promising, and has led to interesting results. The main idea of BUGFIX is to construct all possible paths using correct and incorrect rules between the exercise and the students' solution. A possible disadvantage of the approach is that there are already around 350 buggy rules just for the domain of expressions over integers with addition, subtraction, multiplication, and division. Determining and providing these buggy rules to an e-learning system thus becomes a task that may easily take months. In principle, the third approach suffers from fewer problems. Since feedback is given at and about intermediate steps, fewer buggy rules are applicable compared with the situation where a final answer is compared with the initial exercise. However, most tools do not give more feedback than incorrect/correct. If a student follows a wrong strategy, no

feedback is given. Only ActiveMath gives feedback on the level of strategies, but this feedback seems to have to be specified per exercise. Furthermore, it is impossible to specify a strategy in ActiveMath [1], so adding a new strategy to solve a problem often requires a partial reimplementation of the tool. Furthermore, the tools we have seen do not take the current expression of the student properly into account when providing feedback. Since it is impossible to make errors in rewriting in Math(X)Pert, the feedback given by Math(X)Pert is on the level of strategy. Exactly how feedback works in Math(X)Pert is not documented, but we did find that it is easy to stray away from a good path towards a solution, and that it is possible to confuse Math(X)Pert by doing so.

It is rather unfortunate that considering feedback, current e-learning systems hardly improve upon, and usually are worse than, the 35 year old Goldberg tuition program [16,15]. Goldberg's program is a computer-assisted instruction program that helps students learn Group Theory. Goldberg used a theorem prover to give hints to students when asked for help. A strategy is similar to a 'proof plan' of a theorem prover, so we expect we can provide similar feedback as Goldberg. Already in 1983, Bundy [8] says: 'Goldberg's system is based on the tenet that a teacher must understand something if (s)he is to teach it successfully, even if the teacher is a computer program. This may seem obvious, but it is a tenet which is violated by the more conventional 'drill and practice' Computer Aided Instruction programs.' The situation in 2007 is not much different from the situation in 1983, or 1973. The fundamental problem is that in most e-learning systems, strategies for solving problems are not explicitly modelled. Without explicit strategies it becomes difficult to reason about strategies, and to provide feedback. To quote Bundy [8] again: 'Whatever aspect of intelligence you attempt to model in a computer program, the same needs arise over and over again

- The need to have knowledge about the domain.
- The need to reason with that knowledge.
- The need for knowledge about how to direct or guide that reasoning.'

Note that the three components from which we want to automatically construct feedback: domain, rewrite rules, and strategies, directly correspond to Bundy's essential components of intelligence. These three components are essential for proper mathematical knowledge management [12] when it comes to modeling mathematical exercises.

Many existing e-learning tools for mathematical exercises use Computer Algebra Systems (CAS) to verify correctness of answers. This is attractive, because mathematical knowledge present in a CAS is reused for the e-learning tool. However, reusing a CAS in an e-learning tool has a distinctive disadvantage that the strategy for solving a mathematical problem is not directly available to the e-learning tool, and hence it is impossible to give feedback about possible misunderstandings of the strategy. Furthermore, the strategy for solving a mathematical problem used by a CAS might differ from the strategy the student should learn.

3 An example domain

This section gives an example in which we specify the three components we need for automatically constructing feedback. It introduces the domain of classical logic expressions, the rules for logical expressions, and the strategy for rewriting a logical expression to normal form.

The domain. An example of a logical expression is $\neg(x \vee (y \wedge z))$. A logical expression is a logical variable, a constant `true` or `false`, the negation of a logical expression, or the conjunction, disjunction, or implication of two logical expressions. In a grammar:

$$\begin{aligned} \text{Logic} ::= {} & \text{Var} \\ \mid {} & \texttt{true} \\ \mid {} & \texttt{false} \\ \mid {} & \neg\text{Logic} \\ \mid {} & \text{Logic} \wedge \text{Logic} \\ \mid {} & \text{Logic} \vee \text{Logic} \\ \mid {} & \text{Logic} \rightarrow \text{Logic} \end{aligned}$$

An identifier starting with a capital is a non-terminal, and a lower-case identifier is a terminal.

The rules. Logical expressions form a boolean algebra, and hence there exist a number of rules for logical expressions, such as `true` is the unit of $\wedge$, `false` is the zero of $\wedge$, and $\wedge$ is commutative and associative. Each rule is given a name.

$$\begin{array}{ll} \textsc{TrueLeftUnitAnd}: & \texttt{true} \wedge x = x \\ \textsc{FalseLeftZeroAnd}: & \texttt{false} \wedge x = \texttt{false} \\ \textsc{AndComm}: & x \wedge y = y \wedge x \\ \textsc{AndLeftAssoc}: & x \wedge (y \wedge z) = (x \wedge y) \wedge z \end{array}$$

where x, y, and z range over any logical expression. Similar rules hold for $\vee$. For negation we have de Morgan's rules, amongst others.

$$\begin{array}{ll} \textsc{NotTrue}: & \neg\texttt{true} = \texttt{false} \\ \textsc{NotFalse}: & \neg\texttt{false} = \texttt{true} \\ \textsc{NotNot}: & \neg\neg x = x \\ \textsc{DeMorganAnd}: & \neg(x \wedge y) = \neg x \vee \neg y \\ \textsc{DeMorganOr}: & \neg(x \vee y) = \neg x \wedge \neg y \end{array}$$

And, finally, $\vee$ and $\wedge$ distribute over each other.

$$\begin{array}{l} \textsc{AndRightOverOr}: (x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z) \\ \textsc{OrRightOverAnd}: (x \wedge y) \vee z = (x \vee z) \wedge (y \vee z) \end{array}$$

Buggy rules Many buggy rules can be formulated for the domain of logical expressions. For example, a student might forget to change an $\wedge$ into an $\vee$ in the De Morgan rules:

$$\begin{array}{ll} \textsc{DeMorganAndBuggy}: & \neg(x \wedge y) = \neg x \wedge \neg y \\ \textsc{DeMorganOrBuggy}: & \neg(x \vee y) = \neg x \vee \neg y \end{array}$$

A strategy. To rewrite a logical expression to normal form, the basic rules for logical expressions have to be combined to describe how a logical expression is rewritten to disjunctive normal form. One possible strategy for rewriting logical formulas to disjunctive normal form is to

- first eliminate all $\neg$'s that are not in front of an expression variable by means of any of the rules for $\neg$.
- Then bottom-up eliminate all $\vee$'s that appear below top level, using the rule that says that $\wedge$ distributes over $\vee$.

Both of these two parts have to be applied until they cannot be applied anymore. From this informal description we see that to specify a strategy, we need at least the concepts of

- applying a single basic rewrite rule ('using the rule that $\wedge$ distributes over $\vee$'),
- choice ('any of the rules for $\neg$', denoted by $|$),
- sequence ('first ... Then ...', denoted by ;),
- repeat until exhausted ('until they cannot be applied anymore', denoted by `repeat`),
- bottom-up (and top-down and anywhere, denoted by `bottomUp`).

If we have these concepts available, we can specify a strategy for rewriting a logical expression to normal form as follows:

```
Dnf          = EliminateNots ; MoveOrToTop
EliminateNots = repeat (DeMorganAnd | DeMorganOr | NotNot
                      | NotTrue | NotFalse)
MoveOrToTop  = repeat
                  (bottomUp (AndLeftOverOr | AndRightOverOr))
```

For example, if we apply this strategy to the example logical expression, we get the following derivation:

$$\begin{array}{ll} & \neg(x \vee (y \wedge z)) \\ = & \quad \textsc{DeMorganOr} \\ & \neg x \wedge \neg(y \wedge z) \\ = & \quad \textsc{DeMorganAnd} \\ & \neg x \wedge (\neg y \vee \neg z) \\ = & \quad \textsc{AndLeftOverOr} \\ & (\neg x \wedge \neg y) \vee (\neg x \wedge \neg z) \end{array}$$

If we view Dnf as the specification of a language of transformation steps, the three-rule sequence [DeMorganOr, DeMorganAnd, AndLeftOverOr] can be viewed as a sentence from the strategy Dnf transforming $\neg(x \vee (y \wedge z))$ into its disjunctive normal form.

4 Specifying strategies

Strategies are usually specified informally. To be able to automatically construct feedback for a student, we have to make the strategy that has to be used explicit. How do we specify a strategy? A strategy determines how basic steps are combined together to reach a solution to a problem. An example of an explicit strategy has been given in the previous section. This section discusses a formalism for specifying strategies.

Concepts like applying a basic rewrite rule, choice, sequence, etc. all appear in a strategy language like Stratego, and a Stratego-like language for specifying strategies seems feasible [35,34,25]. We use the following grammar for specifying strategies:

```
Strategy ::= Var
           | basic Rule
           | Strategy | Strategy
           | Strategy ; Strategy
           | repeat Strategy
           | bottomUp Strategy
```

Var generates names that can be used in strategies, and Rule generates the set of basic rewrite rules that may be applied in a particular strategy. An example of a strategy, a sentence of this language of strategies, is the strategy Dnf defined in the previous section.

This is a rather basic and incomplete definition of strategies; the following components are missing:

- Sometimes we want to specify that it is possible to work on different parts of a problem separately, so we need a parallel strategy combinator.
- Some basic rules require input from the student. For example, when subtracting two equations we need a multiplier for one of those equations. So we need variants of the `basic` combinator, with which we can specify that student input is required. Other basic rules require multiple selections in the domain. For example, when substituting $x = \ldots$ in another equation, we have to select both the equation to be substituted, and the equations in which it is to be substituted. Again we need a variant of the `basic` combinator to specify how many selections in the domain have to be made.
- Some exercises require the student to show the existence of an object that satisfies a particular property. So the student has to supply a particular

value. We expect we have to add a combinator to the strategy specification language with which we can express that a student has to supply a value at a particular point in a strategy.

We have experimented with specifying strategies in the domains of logic and parts of linear algebra. However, we have to experiment with more domains to validate the choice of language for specifying strategies. We need a language with at least the power of a context-free grammar, since we want to be able to specify strategies that require performing a certain substrategy n times, and then another substrategy equally often. Such a strategy cannot be expressed by means of a regular grammar (or a finite-state automaton).

5 Feedback on strategies

Our main reason to explicitly specify a strategy is to give feedback to a student if (s)he does not follow the strategy. A second reason is to be able to give a hint when a student asks for it. Using techniques from the field of parser generators, parser combinators and error-correcting parsers, we can automatically construct feedback if a step submitted by the student is not valid according to the strategy. This section explains how we do this.

To determine whether or not a student is on the right track in solving an exercise using a particular strategy, we try to determine whether or not the sequence of transformations specified by the student is a prefix (an initial segment) of a sentence from the grammar specified by the strategy.

We have implemented a set of combinators corresponding to the strategy combinators specified in Section 4 in Haskell [30]. These combinators are very similar to the parser combinators used in higher-order lazy functional programming [21,32]. But instead of parsing sentences, they recognize initial segments of sentences. For example, we can write (`<*>` corresponds to ;, `<|>` to |):

```
dnf  =  eliminateNots <*> moveOrToTop

eliminateNots  =  repeatExhausted
  (basic DeMorganAnd <|> basic DeMorganOr  <|> basic NotNot
 <|> basic NotTrue <|> basic NotFalse)

moveOrToTop   =  repeatExhausted
  (bottomUp (basic AndLeftOverOr <|> basic AndRightOverOr))
```

Here, the `<*>`-combinator takes two recognizers, and tries to recognize the first followed by the second. The `<|>`-combinator takes two recognizers, and tries to either recognize the first or the second. The `basic`-combinator recognizes a single transformation step. To check whether or not a transformation step can be applied, we have to know *where* it has to be applied in an expression. For example, the transformation step `DeMorganAnd` is only applicable to the right argument of the operator $\wedge$ in the expression $\neg x \wedge \neg(y \wedge z)$. It follows that a

transformation step has to be accompanied by a selection, which tells the system where to apply a transformation step. So each `basic` recognizer recognizes a transformation step, followed by a selection in the expression where the transformation step has to be applied. We do not describe the implementation of the library in this paper; the implementation will be published with this paper on our publications page[3].

Recognizers have an underlying state: the expression (exercise) that is being transformed. Each basic rule represents a rewrite step. So besides combinators for recognizers, we also need an implementation for the rules. At the moment we use Strafunski [23], a strategy rewriting library for Haskell, for this purpose. For example, the rule `DeMorganAnd` is implemented as follows:

```
ruleDeMorganAnd (Not (x :&&: y))  =  return (Not x :||: Not y)
ruleDeMorganAnd x                 =  fail "DeMorganAnd"
```

Here `Not` is the ASCII representation of $\neg$, `:&&:` of $\wedge$, and `:||:` of $\vee$. Since the underlying language of Strafunski is Haskell, we can encode powerful rewrite rules with side-conditions, if-then-else expressions, etc. For example, for arithmetical expressions we would define amongst others:

```
ruleDiv (Con x :/: Con 0)  =  fail "Divide by zero"
ruleDiv (Con x :/: Con y)  =  return (x/y)
ruleDiv x                  =  fail "Div"
```

Each time we recognize a transformation step we apply the corresponding rewrite rule to the underlying state.

The standard approach to recognizers would give a program that either says accept or reject. We want to provide better feedback than just correct/incorrect. For this purpose, we use a variant of *error-correcting* parsers [33]. In particular, we have used the 'polish parsers' introduced by Swierstra and Hughes [20] to rewrite our combinators into recognizers that provide better feedback. For example, if a student submits the sequence (where submitting a sequence might mean pressing the buttons corresponding to the transformation steps, and selecting subexpressions) [DeMorganOr, DeMorganAnd] together with the appropriate selection commands, to rewrite $\neg(x \vee (y \wedge z))$, and then says she is finished, our tool signals that a transformation step is missing, namely AndLeftOverOr. How to report this to the student is a different question ('the solution of the exercise has not been reached yet', or, 'you should try to apply the rule that distributes $\wedge$ over $\vee$', or ...). The adapted recognizer analyzes the strategy, signals errors, and suggests possible correct steps when asked for.

When using error-correcting parsers, the feedback is always given on the level of a single transformation step. It is desirable to provide feedback at a higher level. For example, when rewriting $\neg(x \vee (y \wedge (z \vee w)))$ with the sequence [DeMorganOr, AndLeftOverOr] we might prefer the error message 'you have not eliminated all $\neg$'s not in front of a variable yet' instead of a message

[3] See `http://www.cs.uu.nl/~johanj/publications`

'insert the transformation-step DeMorganAnd'. This kind of high-level feedback can be specified in the recognizer. For example, for the `dnf` strategy we can write

```
eliminateNotM  =  "not all not's in front of a variable have been
                   eliminated yet"
moveOrToTopM   =  "not all or's have been moved to top level yet"

dnf  =  addMessage eliminateNotM eliminateNots
    <*> addMessage moveOrToTopM moveOrToTop
```

Using this recognizer, we get the desired feedback in the above example.

6 Conclusions

We have shown how we can give good feedback to students interactively solving an exercise on the level of the strategy for solving the exercise. As far as we are aware, our approach to automatically constructing feedback is novel.

Good feedback can only be given if the domain, the rules with which an expression in the domain can be rewritten, possibly known buggy rules, and a strategy for solving an exercise in the domain are explicit.

We have introduced a context-free grammar like formalism for specifying strategies. A strategy specified in this formalism can be viewed as a grammar. When solving an exercise, a student constructs (prefixes of) a sentence of this grammar. By using techniques from the field of parser generators, parser combinators, and error-correcting parsers, we have implemented a library, with which we can build recognizers for particular strategies which recognize prefixes of sentences of a grammar, and which can automatically construct feedback if a step submitted by the student is not valid according to the strategy.

Using our results, we can build an e-learning system that provides good feedback. There are many advantages of such a system:

- Feedback may be given at each step in a calculation towards a solution, not just after solving an exercise. Thus feedback is given earlier than in tools that construct feedback based on a final answer, and almost always more accurate.
- Feedback is automatically constructed for each intermediate step for each exercise. Feedback need not be specified for every exercise anymore, and hence defining new exercises becomes much easier.
- Feedback is automatically constructed for each strategy. Hence, feedback need not be specified for every class of exercises anymore, and defining a strategy for a new class of problems, or a new strategy for solving a particular class of problems does not require developing or reimplementing a tool.
- To construct an e-learning system for a new domain is a matter of specifying the domain, the rules, the buggy rules, and the strategies. The e-learning system comes for free. A single framework for constructing e-learning systems for different domains suffices. We think it is not realistic to assume

that teachers can adapt e-learning tools to cater for new strategies. We do think there is a possibility that teachers can adapt a strategy to their needs. Thus an e-learning system that is generated from a domain, its rules, and a strategy makes it easier for a teacher to adapt an e-learning tool. This probably increases the acceptance and usage of such a tool by teachers.

Future work. We have only taken the first steps towards an e-learning system that provides good feedback. We have to integrate the results from this paper into our existing e-learning systems to experiment with the expressiveness of our formalism. We have already formulated a number of strategies as recognizers, but we have to experiment with a larger set of strategies to validate the formalism introduced in this paper. Some problems we want to work on in the future are:

- To allow working on different parts of an exercise using a different strategy, we want to introduce a parallel combinator. We expect we can use variants of permutation parsers [2] to implement a parallel combinator.
- How do we deal with strategies in which many trivial intermediate steps are silently applied (such as associativity and commutativity of $\vee$)?
- Can we also specify 'refinement' rules instead of transformation rules? Refinement rules are useful when developing a program or a UML model, for example.
- Some exercises require the student to show the existence of an object that satisfies a particular property. So the student has to supply a particular value. We expect we have to add a combinator to the strategy specification language with which we can express that a student has to supply a value at a particular point in a strategy.
- How do we deal with buggy strategies, or with sub-optimal strategies? We expect we will add 'known' buggy strategies to strategy descriptions, and build recognizers to recognize such buggy strategies, and provide appropriate error messages. Dealing with sub-optimal strategies might be done similarly: recognize and comment. The difference might be that we do not allow a student to make progress when in a buggy strategy, whereas we allow a student to follow a sub-optimal strategy.
- We expect we will need several variants of the `basic` combinator. At the moment the `basic` combinator may be preceded by a number of selection commands, which determine where the transformation step is applied in the underlying expression. However, some transformation steps need more than one selection in the underlying expression. For example, substitution requires selecting an equation of the form $x = \ldots$, and a set of equations in which this equation is substituted. Other transformation steps require student input, which might also be considered as a kind of selection.
- Using the strategy descriptions given in this paper, we can give good feedback when a student makes an error. Giving a hint when a student is stuck can be done using almost the same approach, with the exception that we have to be specific about which choice to make when a student has to provide input to a transformation step. For example, a student sometimes has to

supply a particular value for a variable. We expect we have to adapt our strategy language such that transformation steps that require student input also contain information about how to choose appropriate input when such input is not given, or when a student asks for a hint.

Acknowledgements. Doaitse Swierstra helped in implementing the recognizers for strategies. Eric Bouwers spent a lot of time in finding related work. Josje Lodder and Arthur van Leeuwen commented on a previous version of this paper. The partners in the Dutch Surf educational innovation project Intelligent Feedback in e-learning, in particular Erik Jansen, Hans Cuypers, and Evert van de Vrie, stimulated the research reported in this paper.

References

1. The Activemath Group, Erica Melis, Jochen Büdenbender, George Goguadze, Paul Libbrecht, and Carsten Ullrich. Knowledge representation and management in activemath. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):47–64, 2003.
2. Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004.
3. Michael J. Beeson. A computerized environment for learning algebra, trigonometry, and calculus. *Journal of Artificial Intelligence and Education*, 1:65–76, 1990.
4. Michael J. Beeson. Design principles of Mathpert: Software to support education in algebra and calculus. In N. Kajler, editor, *Computer-Human Interaction in Symbolic Computation*, pages 89–115. Springer-Verlag, 1998.
5. C Bokhove, A. Heck, and G. Koolstra. Intelligent feedback to digital assessments and exercises (in Dutch). *Euclides*, 2, 2005.
6. John Seely Brown and Richard R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2:155–192, 1978.
7. John Seely Brown and Kurt VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.
8. Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.
9. H. Chaachoua et al. Aplusix, a learning environment for algebra, actual use and benefits. In *ICME 10 : 10th International Congress on Mathematical Education*, 2004. Retrieved from `http://www.itd.cnr.it/telma/papers.php`, January 2007.
10. Arjeh Cohen, Hans Cuypers, Dorina Jibetean, and Mark Spanbroek. Interactive learning and mathematical calculus. In *Mathematical Knowledge Management*, 2005.
11. Hans van Ditmarsch. Logic software and logic education. `http://www.cs.otago.ac.nz/staffpriv/hans/logiccourseware.html`. These pages contain a comprehensive, alphabetically ordered list of educational logic software, 2005.
12. William M. Farmer. MKM: a new interdisciplinary field of research. *SIGSAM Bull.*, 38(2):47–52, 2004.
13. Freudenthal Institute. Digital math environment. `http://www.fi.uu.nl/dwo`, 2004.
14. G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. In *International Conference on Computers in Education, ICCE 2005*, 2005.

15. A. Goldberg and P. Suppes. A Computer-Assisted Instruction Program for Exercises on Finding Axioms. *Educational Studies in Mathematics*, 4:429–449, 1972.
16. Adele Goldberg. *Computer assisted instruction: The application of theorem proving to adaptive response analysis*. PhD thesis, Stanford, May 1973.
17. Ira P. Goldstein and Eric Grimson. Annotated production systems: A model for skill acquisition. In *IJCAI*, pages 311–317, 1977.
18. André Heck and Leendert van Gastel. Diagnostic testing with Maple T.A. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 37–52. Oy WebALT Inc., 2006.
19. Martin Hennecke. *Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen (in German)*. PhD thesis, Hildesheim University, 1999. Fortschritt-Berichte VDI Reihe 10, Informatik / Kommunikationstechnik; 605. Düsseldorf: VDI-Verlag.
20. R. John M. Hughes and S. Doaitse Swierstra. Polish parsers, step by step. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 239–248. ACM Press, 2003.
21. Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.
22. Anan Erev Ido Erev, Adi Luria. On the effect of immediate feedback, May 2006. Retrieved from `http://telem-pub.openu.ac.il/users/chais/2006/05/pdf/e-chais-erev.pdf`, December 2006.
23. Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 137–154. Springer-Verlag, 2002.
24. Josje Lodder, Johan Jeuring, and Harrie Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.
25. Mircea Marin and Tetsuo Ida. Progress of rholog, a rule-based programming system. *Mathematica in Education and Research*, 11(1):50–66, 2006.
26. Manolis Marvrikis and Antony Macioncia. Wallis: a web-based ILE for science and engineering students studying mathematics. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 111–126. Oy WebALT Inc., 2006.
27. B. Jean Mason and Roger Bruning. Providing feedback in computer-based instruction: What the research tells us. retrieved from `http://dwb.unl.edu/Edit/MB/MasonBruning.html`, august 2006, 2001.
28. E. H. Mory. Feedback research revisited. In D. H. Jonassen, editor, *Handbook of Research on Educational Communications and Technology*, chapter 29, pages 745–783. Lawrence Erlbaum Associates, 2004.
29. Harrie Passier and Johan Jeuring. Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc., 2006.
30. Simon 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, see also `http://www.haskell.org/`.
31. Christopher J. Sangwin and Michael Grove. Stack: addressing the needs of the ¨neglected learners¨. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 81–96. Oy WebALT Inc., 2006.

32. Doaitse Swierstra. Combinator parsers: From toys to tools. In *Haskell Workshop 2000*, September 2000.
33. S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.
34. Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.
35. Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.