Strategies for Exercises

Bastiaan Heeren

Johan Jeuring

Alex Gerdes

Technical Report UU-CS-2009-003
January 2009

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

Strategies for Exercises

Bastiaan Heeren, Johan Jeuring and Alex Gerdes

Abstract. Strategies specify how a wide range of exercises can be solved incrementally, such as bringing a logic proposition to disjunctive normal form, reducing a matrix, or calculating with fractions. In this paper we introduce a language for specifying strategies for solving exercises. This language makes it easier to automatically calculate feedback, for example when a user makes an erroneous step in a calculation. We can automatically generate worked-out solutions, track the progress of a student by inspecting submitted intermediate answers, and report back suggestions in case the student deviates from the strategy. Thus it becomes less labor-intensive and less ad-hoc to specify new exercise domains and exercises within that domain.

A strategy describes valid sequences of transformation rules that solve the exercise at hand, which turns tracking intermediate steps into a parsing problem. This is a promising view at the problem because it allows us to take advantage of many years of experience in parsing sentences of context-free languages, and transfer this knowledge and technology to the domain of stepwise solving exercises.

In this paper we work out the similarities between parsing and solving exercises incrementally, and we discuss the implementation of a recognizer for strategies. We present a full implementation of such a recognizer, and discuss a number of design choices we have made. In particular, we discuss the use of a fixed point combinator to deal with repetition, and labels to mark positions in the strategy.

1. Introduction

Tools like Aplusix [9], ActiveMath [15], MathPert [4], and our own tool for rewriting logic expressions [25] support solving mathematical exercises incrementally. Ideally a tool gives detailed feedback on several levels. For example, when a student rewrites $p \rightarrow (r \leftrightarrow p)$ into $\neg p \vee (r \leftrightarrow p$, our tool will tell the student that there is a missing parenthesis. If the same expression is rewritten into $\neg p \wedge (r \leftrightarrow p)$, it

will tell the student that an error has been made when applying the definition of implication: correct application of this definition would give $\neg p \vee (r \leftrightarrow p)$. Finally, if the student rewrites $\neg(p \wedge (q \vee r))$ into $\neg((p \wedge q) \vee (p \wedge r))$, it will tell the student that although this step is not wrong, it is better to first eliminate occurrences of $\neg$ occurring at top-level, since this generally leads to fewer rewrite steps.

The first kind of error is a syntax error, and there exist good error-repairing parsers that suggest corrections to formulas with syntax errors. The second kind of error is a rewriting error: the student rewrites an expression using a non-existing or buggy rule. There already exist some interesting techniques for finding the most likely error when a student incorrectly rewrites an expression. The third kind of error is an error on the level of the procedural skill or strategy for solving this kind of exercises. This paper discusses how we can formulate and use strategies to construct the third kind of feedback.

The main contribution of this paper is the formulation of a strategy language as a domain-specific embedded language, with a clear separation between a context-free and a non-context-free part. The strategy language can be used for any domain, and can be used to automatically calculate feedback on the level of strategies, given an exercise, the strategy for solving the exercise, and student input. Another contribution of our work is that the specification of a strategy and the calculation of feedback is separated: we can use the same strategy specification to calculate different kinds of feedback. Furthermore, we discuss the implementation of a recognizer for such strategies, and discuss a number of design choices we have made. In particular, we discuss the use of a fixed point combinator to deal with repetition, and labels to mark positions in the strategy.

This paper is organized as follows. Section 2 introduces strategies, and discusses how they can help to improve feedback in e-learning systems or intelligent tutoring systems. We continue with some example strategies from the domain of logical expressions (Section 3). Then, we present our language for writing strategies in Section 4. We do so by defining a number of strategy combinators, and by showing how the various example strategies can be specified in our language. Section 5 shows how we have implemented the components of our strategy language to obtain a strategy recognizer, and discusses the main design choices. Section 6 discusses several possibilities for giving feedback or hints using our strategy language. The last section concludes and gives directions for future research.

2. Strategies and feedback

Whatever aspect of intelligence you attempt to model in a computer program, the same needs arise over and over again [8]:

- 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.

In the case of exercises, the knowledge about how to guide reasoning is often captured by a so-called procedure or procedural skill. 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.

Many subjects require a student to learn strategies. At elementary school, students have to learn how to calculate a value of an expression, which may include fractions. At high school, students learn how to solve a system of linear equations, and at university, students learn how to apply Gaussian elimination to a matrix, or how to rewrite a logical expression to disjunctive normal form (DNF). Strategies are not only important for mathematics, logic, and computer science, but also for physics, biology (Mendel's laws), and many other subjects. Strategies are taught at any level, in almost any subject, and range from simple – for example the simplification of arithmetic expressions – to very complex – for example a complicated linear algebra procedure.

E-learning systems for learning strategies. Strategic skills are almost always acquired by practicing exercises, and indeed, students usually equate mathematics with solving exercises. In schools, the dominant practice still is a student performing a calculation using pen-and-paper, and the teacher correcting the calculation (the same day, in a couple of days, after a couple of weeks). There exist many software solutions that support practicing solving exercises on a computer. The simplest kinds of tools offer multiple-choice questions, possibly with an explanation of the error if a wrong choice is submitted. A second class of tools asks for an answer to a question, again, possibly with an analysis of the answer to give feedback when an error has been made. The class of tools we consider in the paper are tools that support the incremental, step-wise calculation of a solution to an exercise, thus mimicking the pen-and-paper approach more or less faithfully. Since e-learning tools for practicing procedural skills seem to offer many advantages, hundreds of tools that support practicing strategies in mathematics, logic, physics, etc. have been developed.

Should e-learning systems give feedback? In Rules of the Mind [1], Anderson discusses the ACT-R principles of tutoring, and the effectiveness of feedback in intelligent tutoring systems. One of the tutoring principles deals with student errors. If a student made a slip in performing a step (s)he should be allowed to correct it without further assistance. However, if a student needs to learn the correct rule, the system should give a series of hints with increasing detail, or show how to apply the correct rule. Finally, it should also be possible to give an explanation of an error made by the student. The question on whether or not to give immediate feedback is still debated. Anderson observed no positive effects in learning with

deferred feedback, but observed a decline in learning rate instead. Erev et al. [13] also claim that immediate feedback is often to be preferred.

Feedback in e-learning systems supporting incrementally solving exercises. There are only very few tools that mimic the incremental pen-and-paper approach and that give feedback at intermediate steps different from correct/incorrect. Although the correct/incorrect feedback at intermediate steps is valuable, it is unfortunate that the full possibilities of e-learning tools are not used. There are several reasons why the given feedback is limited. The main reasons probably are that supporting detailed feedback for each exercise is very laborious, providing a comprehensive set of possible bugs for a particular domain requires a lot of research (see for example Hennecke's work [17] on student bugs in calculating fractions), and automatically calculating feedback for a given exercise, strategy, and student input is very difficult.

Feedback should be calculated automatically. We think specifying feedback together with every exercise that is solved incrementally is a dead-end: teachers will want to enter new exercises on a regular basis, and completely specifying feedback is just too laborious, error prone, and repetitive. Instead, feedback should in general be calculated automatically, given the exercise, the strategy for the exercise, buggy rules and strategies, and the input from the student. To automatically calculate feedback, we need information about the domain of the exercise, the rules for manipulating expressions in this domain, the strategy for solving the exercise, and common bugs. For example, for Gaussian elimination of a matrix, we have to know about matrices (which can be represented by a list of rows), the rules for manipulating matrices (the elementary matrix operations such as scaling a row, subtracting a row from another row, and swapping two rows), buggy rules and strategies for manipulating matrices (subtracting a row from itself), and the strategy for Gaussian elimination of a matrix, which can be found in the technical report corresponding to this paper [16].

Representing strategies. Representing the domain and the rules for manipulating an expression in the domain is often relatively straightforward. Specifying a strategy for an exercise is more challenging in many cases. To specify a strategy, we need the power of a full programming language: many strategies require computations of values. However, to calculate feedback based on a strategy, we need to know more than that it is a program. We need to know its structure and basic components, which we can use to report back on errors. Furthermore, we claim that if we ask a teacher to write a strategy as a program, instead of specifying feedback with every exercise, the automatic approach is not going to be very successful.

An embedded domain-specific language for specifying strategies. This paper discusses the design of a language for specifying strategies for exercises. The domains and rules vary for the different subjects, but the basic constructs for describing strategies are the same for different subjects ('first do this, then do that', 'either do

this or that'). So, the strategy language can be used for any domain (mathematics, logic, physics, etc). It consists of several basic constructs from which strategies can be built. These basic constructs are combined with program code in a programming language to be able to specify any strategy. The strategy language is formulated as an embedded domain-specific language (EDSL) in a programming language [20] to easily facilitate the combination of program code with a strategy. Here 'domain-specific' means specific for the domain of strategies, not specific for the domain of exercises. The separation into basic strategy constructs and program code offers us the possibility to analyze the basic constructs, from which we can derive several kinds of feedback.

What kind of feedback? We can automatically calculate the following kinds of feedback, many of which are part of the tutoring principles of Anderson [1].

- Is the student still on the right path towards a solution? Does the step made by the student follow the strategy for the exercise? What is the next step the student should take?
- We produce hints based on the strategy.
- We can give an indication of the progress, based on the position on the path from the starting point to the solution of an exercise.
- If a student enters a wrong final answer, we can ask the student to solve subproblems of the original problem.
- We allow the formulation of buggy strategies to explain common mistakes to students.

We do not build a model of the student to try to explain the error made by the student. According to Anderson, an informative error message is better than bug diagnosis.

How do we calculate feedback on strategies? The strategy language is defined as an embedded domain-specific language in Haskell [28]. Using the basic constructs from the strategy language, we can create something that resembles a context-free grammar. The sentences of this grammar are sequences of transformation steps (applications of rules). We can thus check whether or not a student follows a strategy by parsing the sequence of transformation steps, and checking that the sequence of transformation steps is a prefix of a correct sentence from the context-free grammar. Many steps require student input, for example when a student wants to multiply a row by a scalar, or when a student wants to subtract two rows. This part of the transformation cannot be checked by means of a context-free grammar, and here we make use of the fact that our language is embedded into a full programming language, to check input values supplied by the student. The separation of the strategy into a context-free part, using the basic strategy combinators, and a non-context-free part, using the power of the programming language, offers us the possibility to give the kinds of feedback mentioned above. Computer Science has almost 50 years of experience in parsing sentences of context-free languages,

including error-repairing parsers, which we can use to improve feedback on the level of strategies.

Related work. Explaining syntax errors has been studied in several contexts, most notably in compiler construction [31], but also for e-learning tools [19]. Some work has been done on trying to explain errors made by students on the level of rewrite rules [17, 22, 26, 5].

Already around 1980, but also later, VanLehn et al. [6, 7, 32], and Anderson and others from the Advanced Computer Tutoring research group at CMU [1, 2] worked on representing procedures or procedural networks. VanLehn et al. already noticed that 'The representation of procedures has an impact on all parts of the theory.' Anderson et al. report that the technical accomplishment was 'no mean feat'. Both VanLehn et al. and Anderson et al. chose to deploy collections of condition-action rules, or production systems. In Mind Bugs [32], VanLehn states several assumptions about languages for representing procedures. In Rules of the Mind [1], Anderson formulates similar assumptions. Their leading argument for selecting a language for representing procedures is that it should be psychologically plausible. We think our strategy language can be viewed as a production system. But our leading argument is that it should be easy to calculate feedback based on the strategy. Using an EDSL for specifying the context-free part of a strategy simplifies calculating feedback. Furthermore, our language satisfies the assumptions about representation languages given by VanLehn, such as the presence of variables in procedures, and the possibility to define recursive procedures. As far as we can see, neither VanLehn nor Anderson use parsing for the language for procedures to automatically calculate feedback.

Zinn [34] writes strategies as Prolog programs, in which rules and strategies ('task models') are intertwined.

3. Three example strategies

In this section we present three strategies for rewriting a classical logical expression to disjunctive normal form. Although the example strategies are relatively simple, they are sufficiently rich to demonstrate the main components of our strategy language.

The domain. Before we can define a strategy, we first have to introduce the domain of logical expressions and a collection of available rules. A logical expression is a logical variable, a constant *true* or *false* (written T and F), the negation of a logical expression, or the conjunction, disjunction, implication, or equivalence of two logical expressions. This results in the following grammar:

Basic Rules:		
Constants:	AndTrue: $p \wedge T = p$	AndFalse: $p \wedge F = F$
	OrTrue: $p \vee T = T$	OrFalse: $p \vee F = p$
	NotTrue: $\neg T = F$	NotFalse: $\neg F = T$
Definitions:	ImplDef: $p \rightarrow q = \neg p \vee q$	
	EquivDef: $p \leftrightarrow q = (p \wedge q) \vee (\neg p \wedge \neg q)$	
Negations:	NotNot: $\neg\neg p = p$	
	DeMorganAnd: $\neg(p \wedge q) = \neg p \vee \neg q$	
	DeMorganOr: $\neg(p \vee q) = \neg p \wedge \neg q$	
Distribution:	AndOverOr: $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	
Additional Rules:		
Tautologies:	ImplTaut: $p \rightarrow p = T$	OrTaut: $p \vee \neg p = T$
	EquivTaut: $p \leftrightarrow p = T$	
Contradictions:	AndContr: $p \wedge \neg p = F$	EquivContr: $p \leftrightarrow \neg p = F$

Figure 1: Rules for logical expressions

$$
\begin{array}{rcl}
\mathit{Logic} & ::= & \mathit{Var} \mid T \mid F \mid \neg \mathit{Logic} \mid \mathit{Logic} \wedge \mathit{Logic} \\
& \mid & \mathit{Logic} \vee \mathit{Logic} \mid \mathit{Logic} \rightarrow \mathit{Logic} \mid \mathit{Logic} \leftrightarrow \mathit{Logic} \\
\mathit{Var} & ::= & p \mid q \mid r \mid \ldots
\end{array}
$$

If necessary, we write parentheses to resolve ambiguities. Examples of valid expressions are $\neg(p \vee (q \wedge r))$ and $\neg(\neg p \leftrightarrow p)$.

The rules. Logical expressions form a boolean algebra, and hence a number of rules for logical expressions can be formulated. Figure 1 presents a small collection of basic rules and some tautologies and contradictions. All variables in these rules are meta-variables and range over arbitrary logical expressions. The rules are expressed as equivalences, but are only applied from left to right. For most rules we assume to have a commutative variant, for instance, $T \wedge p = p$ for rule AndTrue. With these implicit rules, we can bring every logical expression to disjunctive normal form.

Every serious exercise assistant for this domain has to be aware of a much richer set of rules. In particular, we have not given rules for commutativity and associativity, several plausible rules for implications and equivalences are omitted, and the list of tautologies and contradictions is far from complete.

Strategy 1: apply rules exhaustively. The first strategy applies the basic rules from Figure 1 exhaustively: we proceed as long as we can apply *some* rule *somewhere*, and we will end up with a logical expression in disjunctive normal form. This is a special property of the chosen collection of basic rules, and this is not the case for

a rule set in general. The strategy is very liberal, and approves every sequence of rules.

Strategy 2: four steps. Strategy 1 accepts sequences that are not very attractive, and that no expert would ever consider. We give two examples:

$$\neg\neg(p \vee q) \overset{\text{DeMorganOr}}{\Longrightarrow} \neg(\neg p \wedge \neg q) \qquad T \vee (\neg\neg p) \overset{\text{NotNot}}{\Longrightarrow} T \vee p$$

In both cases, it is more appealing to select a different rule (NotNot and OrTrue, respectively). We define a new strategy that proceeds in four steps, and such that the above sequences are not permitted.

- **Step 1:** Remove constants from the logical expression with the rules for "constants" (see Figure 1), supplemented with constant rules for implications and equivalences. Apply the rules *top-down*, that is, at the highest possible position in the abstract syntax tree. After this step, all occurrences of T and F are removed.
- **Step 2:** Use ImplDef and EquivDef to rewrite implications and equivalences in the formula. Proceed in a *bottom-up* order.
- **Step 3:** Push negations inside the expression using the rules for negations, and do so in a *top-down* fashion. After this step, all negations appear directly in front of a logical variable.
- **Step 4:** Use the distribution rule (AndOverOr) to move disjunctions to top-level. The order is irrelevant.

Strategy 3: tautologies and contradictions. Suppose that we want to extend Strategy 2, and use rules expressing tautologies and contradictions (for example, the additional rules in Figure 1). These rules introduce constants. Our last strategy is as follows:

- Follow the four steps of Strategy 2, however:
- *Whenever* possible, use the rules for tautologies and contradictions (*top-down*), *and*
- clean up the constants afterwards (step 1). Then continue with Strategy 2.

Buggy rules. In addition to the collection of rules and a strategy, we can formulate *buggy rules*. These rules capture mistakes that are often made, such as the following unsound variations on the two De Morgan rules:

$$\text{BuggyDM1}: \neg(p \wedge q) \neq \neg p \wedge \neg q \qquad \text{BuggyDM2}: \neg(p \vee q) \neq \neg p \vee \neg q$$

The advantage of formulating buggy rules is that specialized feedback can be presented if the system detects that such a rule is applied. Note that these rules should not appear in strategies, since that would invalidate the strategy.

The idea of buggy rules can easily be extended to buggy strategies. A buggy strategy corresponds to a common procedural mistake. Applying a buggy rule

results in an expression with a different semantics from the previous expression. Applying a rule from a buggy strategy results in an equivalent expression, but following a wrong strategy. Also for buggy strategies we want to report a detailed message.

4. A language for strategies for exercises

The previous section gives an intuition of strategies for exercises, such as the three DNF strategies. In this section we define a language for specifying such strategies. We explore a number of combinators to combine simple strategies into more complex ones. We start with a set of basic combinators, and gradually move on to more powerful combinators.

4.1. Basic strategy combinators

Strategies are built on top of basic rules, such as the logic rules from the previous section. Let r be a rule, and let a be some term. We write *apply* r a to denote the application of r to a, which returns a set of terms. If this set is empty, we say that r is not applicable to a, and that the rule fails.

The basic combinators for building strategies are the same as the building blocks for context-free grammars (CFGs). In fact, we can view a strategy as a grammar where the rules form the alphabet of the language.

- **Transformation rule.** Such a rule is the smallest building block to construct composite strategies, and closely corresponds to a terminal symbol in a CFG. Occasionally, we write *symbol* r for some transformation rule r to distinguish the strategy from the transformation rule.
- **Sequence.** Two strategies can be composed and put in sequence. We write $s \mathbin{<\!\star\!>} t$ to denote the sequence of strategy s followed by strategy t.
- **Choice.** We can choose between two strategies, for which we will write $s \mathbin{<|>} t$. One of its argument strategies is applied.
- **Units.** Two special strategies are introduced: *succeed* is the strategy that always succeeds, without doing anything, and *fail* is the strategy that always fails. These combinators are useful to have: *succeed* and *fail* are the unit elements of the $<\!\star\!>$ and $<|>$ combinators.
- **Recursion.** We need a way to deal with recursion, and for this we introduce a fixed point combinator. We write *fix* f, where f is the function of which we take the fixed point. Hence, the function f takes a strategy and returns one, and such that the property *fix* $f = f$ (*fix* f) holds.
- **Labels.** With our final combinator we can label strategies. We write *label* ℓ s to label strategy s with some label ℓ. Labels are used to mark positions in a strategy, and allow us to attach content such as hints and messages to the

strategy. Labeling does not change the language that is generated by the strategy.

We make our strategy combinators precise by defining the semantics of the strategy language. A strategy is a set of sequences of transformation rules.

$$\begin{aligned}
&\mathit{language}\ (s \mathbin{<\!\!*\!\!>} t) &&= \{\ xy \mid x \in \mathit{language}\ s, y \in \mathit{language}\ t\ \}\\
&\mathit{language}\ (s \mathbin{<\!|\!>} t) &&= \mathit{language}\ s \cup \mathit{language}\ t\\
&\mathit{language}\ (\mathit{fix}\ f) &&= \mathit{language}\ (f\ (\mathit{fix}\ f))\\
&\mathit{language}\ (\mathit{label}\ \ell\ s) &&= \mathit{language}\ s\\
&\mathit{language}\ (\mathit{symbol}\ r) &&= \{r\}\\
&\mathit{language}\ \mathit{succeed} &&= \{\epsilon\}\\
&\mathit{language}\ \mathit{fail} &&= \emptyset
\end{aligned}$$

This definition tells us whether a sequence of rules follows a strategy or not: the sequence of rules should be a sentence in the language generated by the strategy, or a prefix of a sentence since we solve exercises incrementally. Not all sequences make sense, however. An exercise gives us an initial term (say t_0), and we are only interested in sequences of rules that can be applied successively to this term. Suppose that we have terms (denoted by t_i) and rules (denoted by r_i), and let t_{i+1} be the result of applying rule r_i to term t_i. A possible derivation that starts with t_0 can be depicted in the following way:

$$t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3 \xrightarrow{r_3} \ldots$$

To be precise, applying a rule to a term can yield multiple results, but most domain rules, such as the rules for logical expressions in Figure 1, return at most one term. Running a strategy s with an initial term t_0 returns a set of terms, and is specified by:

$$\mathit{run}\ s\ t_0 = \{\ t_{n+1} \mid r_0 \ldots r_n \in \mathit{language}\ s, \forall_{i \in 0 \ldots n} : t_{i+1} \in \mathit{apply}\ r_i\ t_i\ \}$$

The rest of this section introduces more strategy combinators to conveniently specify strategies. All these combinators, however, can be defined in terms of the combinators that are given above.

4.2. Extensions

Extended Backus-Naur form (EBNF) extends the notation for grammars, and offers three new constructions that one often encounters in practice: zero or one occurrence (option), zero or more occurrences (closure), and one or more occurrences (positive closure). Along these lines, we introduce three new strategy combinators: *many* s means repeating strategy s zero or more times, with *many1* we have to apply s at least once, and *option* s may or may not apply strategy s. We define these combinators using the basic combinators:

$$
\begin{aligned}
&\mathit{many}\ \ s = \mathit{fix}\ (\lambda x \rightarrow \mathit{succeed} \mathbin{<|>} (s \mathbin{<\star>} x))\\
&\mathit{many1}\ s = s \mathbin{<\star>} \mathit{many}\ s\\
&\mathit{option}\ s = s \mathbin{<|>} \mathit{succeed}
\end{aligned}
$$

Observe the use of the fixed point combinator *fix* in the definition of *many*. Unfolding the *many s* strategy would result in:

$$
\begin{aligned}
&\mathit{many}\ s\\
&\quad = \mathit{succeed} \mathbin{<|>} (s \mathbin{<\star>} \mathit{many}\ s)\\
&\quad = \mathit{succeed} \mathbin{<|>} (s \mathbin{<\star>} (\mathit{succeed} \mathbin{<|>} (s \mathbin{<\star>} \mathit{many}\ s)))\\
&\quad = \ldots
\end{aligned}
$$

It is quite common for an EDSL to introduce a rich set of combinators on top of a (small) set of basic combinators.

4.3. Negation and greedy combinators

The next combinators we consider allow us to specify that a certain strategy is not applicable. Have a look at the definition of *not*, which only succeeds if the argument strategy s is not applicable to the current term a:

$$(\mathit{not}\ s)(a) = \textbf{if}\ s(a) = \emptyset\ \textbf{then}\ \{a\}\ \textbf{else}\ \emptyset$$

Observe that the *not* combinator can be specified as a single rule that either returns a singleton set or the empty set depending on the applicability of strategy s. A more general variant of this combinator is *check*, which receives a predicate as argument (instead of a strategy) for deciding what to return.

Having defined *not*, we now specify greedy variations of *many*, *many1*, and *option* (*repeat*, *repeat1*, and *try*, respectively). These combinators are greedy as they will apply their argument strategies whenever possible.

$$
\begin{aligned}
&\mathit{repeat}\ \ s = \mathit{many}\ s\ \ \mathbin{<\star>}\ \mathit{not}\ s\\
&\mathit{repeat1}\ s = \mathit{many1}\ s \mathbin{<\star>} \mathit{not}\ s\\
&\mathit{try}\ \ \ \ \ s = s \mathbin{<|>} \mathit{not}\ s\\
&s \triangleright t\ \ \ \ \ = s \mathbin{<|>} (\mathit{not}\ s \mathbin{<\star>} t)
\end{aligned}
$$

The last combinator defined, $s \triangleright t$, is a left-biased choice: t is only considered when s is not applicable.

4.4. Traversal combinators

In many domains, terms are constructed from smaller subterms. For instance, a logical expression may have several subexpressions. Because we do not only want to apply rules and strategies to the top-level term, we need some additional combinators to indicate that the strategy or rule at hand should be applied *somewhere*. For this kind of functionality, we need some support from the underlying domain. Let us assume that a function *once* has been defined on a certain domain, which

applies a given strategy to exactly one of the term's immediate children. For the logic domain, this function would contain the following definitions:

$$\begin{aligned} \mathit{once}\ s\ (p \wedge q) &= \{\, p' \wedge q \mid p' \leftarrow \mathit{run}\ s\ p \,\} \cup \{\, p \wedge q' \mid q' \leftarrow \mathit{run}\ s\ q \,\} \\ \mathit{once}\ s\ (\neg p) &= \{\, \neg p' \mid p' \leftarrow \mathit{run}\ s\ p \,\} \\ \mathit{once}\ s\ T &= \emptyset \end{aligned}$$

Using generic programming techniques [18], we can define this function once and for all, and use it for every domain.

With the *once* function, we can define some powerful traversal combinators. The strategy *somewhere s* applies s to one subterm (including the whole term itself).

$$\mathit{somewhere}\ s = \mathit{fix}\ (\lambda x \rightarrow s <|> \mathit{once}\ x)$$

If we want to be more specific about where to apply a strategy, we can instead use *bottomUp* or *topDown*:

$$\begin{aligned} \mathit{bottomUp}\ s &= \mathit{fix}\ (\lambda x \rightarrow \mathit{once}\ x \triangleright s) \\ \mathit{topDown}\ s &= \mathit{fix}\ (\lambda x \rightarrow s \triangleright \mathit{once}\ x) \end{aligned}$$

These combinators search for a suitable location to apply a certain strategy in a bottom-up or top-down fashion, without imposing an order in which the children are visited.

4.5. DNF strategies revisited

In Section 3, we presented three alternative strategies for turning a logical expression into disjunctive normal form. Having defined a set of strategy combinators, we can now give a precise definition of these strategies in terms of our combinators. We start with grouping the rules, as suggested by Figure 1:

$$\begin{aligned} \mathit{basicRules} &= \mathit{constants} <|> \mathit{definitions} <|> \mathit{negations} <|> \mathit{distribution} \\ \mathit{constants} &= \textsc{AndTrue} <|> \textsc{AndFalse} <|> \textsc{OrTrue} <|> \textsc{OrFalse} \\ &\quad <|> \textsc{NotTrue} <|> \textsc{NotFalse} \end{aligned}$$

Definitions for the other groups are similar. The first two strategies can now conveniently be written as:

$$\begin{aligned} \mathit{dnfStrategy1} &= \mathit{repeat}\ (\mathit{somewhere}\ \mathit{basicRules}) \\ \mathit{dnfStrategy2} &= \mathit{label}\ \ell_1\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants})) \\ &\quad <\star> \mathit{label}\ \ell_2\ (\mathit{repeat}\ (\mathit{bottomUp}\ \mathit{definitions})) \\ &\quad <\star> \mathit{label}\ \ell_3\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{negations})) \\ &\quad <\star> \mathit{label}\ \ell_4\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{distribution})) \end{aligned}$$

The labels in the second strategy are not mandatory, but they emphasize the structure of the strategy, and help to attach feedback to this strategy later on. The third strategy can be defined with the combinators introduced thus far, but we give a more elegant definition at the end of the next subsection.

4.6. Parallel strategies

Suppose that we want to run the strategies s and t in parallel, denoted by $s <||> t$. This operator is defined in Section 5.6. This operation makes sense in the domain of rewriting: for example, two parts have to be reduced, and steps to reduce any of the two parts can be interleaved until we are done with both sides. In theory, we can express two strategies that run in parallel in terms of sequences and choices. In practice, however, such a translation does not scale because the grammar will grow tremendously.

The parallel strategy combinator is used in a combinator *whenever*, which takes two strategies s and t, and applies s whenever possible, and t otherwise. It stops when t has been fully applied. Its definition is rather involved, and not presented here. With *whenever* we can give a concise definition for the third DNF strategy, in which we reuse our second strategy.

$$\begin{aligned} \mathit{dnfStrategy3} = \mathit{whenever}\ ((\mathit{tautologies} <|> \mathit{contradictions}) <\star> \mathit{step1}) \\ \mathit{dnfStrategy2} \end{aligned}$$

In the above definition, *step1* is equal to *repeat* (*topDown constants*).

4.7. Reflections

Is the set of strategy combinators complete? Not really, although we hope to have convinced the reader that the language can be easily extended with more combinators. In fact, this is probably the greatest advantage of using an EDSL instead of defining a new, stand-alone language. We believe that our combinators are sufficient for specifying the kind of strategies that are needed in interactive exercise assistants that aim at providing advanced feedback. Our language is very similar to strategic programming languages such as Stratego [33, 24], and very similar languages are used in parser combinator libraries [21, 31], boiler-plate libraries [23], workflow applications [29], theorem proving (tacticals [27]) and data-conversion libraries [12], which suggests that our library could serve as a firm basis for strategy specifications.

Our strategy language can be used to model strategies in multiple mathematical domains. In the technical report corresponding to this paper [16] we present a complete strategy that implements the Gaussian elimination algorithm in the linear algebra domain. This example is more involved than the strategies for DNF, and shows, amongst others, how rules can be parameterised, and how to maintain additional information in a context while running a strategy. In addition, we have strategies for solving a linear system, solving a linear system with a matrix, and applying the Gram-Schmidt algorithm in the linear algebra domain. The report also discusses a strategy for simplifying fractions in the domain of arithmetic expressions. We have also implemented strategies in the domains of relation algebra and calculus (derivatives, higher degree equations).

Producing a strategy is like programming, and might require quite some effort. We think that the effort is related to the complexity of the strategy. Gaussian elimination is an involved strategy, which probably should be written by an expert, but the basic strategy for DNF, *dnfStrategy1*, is rather simple, and could have been written by a teacher of a course in logic. Furthermore, due to compositionality of the strategy combinators, strategies are reusable. In the linear algebra domain, for example, many strategies we have written consist of Gaussian elimination, preceded and followed by some exercise-specific steps (e.g., to find the inverse of a matrix).

We can specify a strategy that is very strict in the order in which steps have to be applied (*dnfStrategy2* enforces a very strict order), or very flexible (*dnfStrategy1* doesn't care about which step is applied when). Furthermore, we can enforce a strategy strictly, or allow a student to deviate from a strategy, as long as the submitted expression is still equivalent to the previous expression, and the strategy can be restarted at that point (this is possible for most of the strategies we have encountered). If we want a student to take clever short-cuts through a strategy, then these shortcuts should either be explicitly specified in the strategy (as in *dnfStrategy3*), or it should be possible to deviate from the given strategy.

5. Design of a strategy recognizer

In this section we discuss the design of a strategy recognizer. Recognizing a strategy comes down to tracing the steps that a student is taking, but how would a tool get the sequence of rules? In exercise assistants that offer free input, users submit intermediate terms. Therefore, the tool first has to determine which of the known rules has been applied, or even which combination of rules has been used. Discovering which rule has been used is obviously an important part of an exercise assistant, and it influences the quality of the generated feedback. It is, however, not the topic of this paper. An alternative to free input is to let users select a rule, which is then applied automatically to the current term. In this setup, it is no longer a problem to detect which rule has been used.

Instead of designing our own recognizer, we could reuse existing parsing libraries and tools. There are many excellent parser generators and various parser combinator libraries around [21, 31], and these are often highly optimized and efficient in both their time and space behavior. However, the problem we are facing is quite different from other parsing applications. To start with, efficiency is not a key concern. Because we are recognizing applications of rewrite rules applied by a student, the length of the input is very limited. Our experience until now is that speed poses no serious constraints on the design of the library. A second difference is that we are not building an abstract syntax tree.

The following issues are important for a strategy recognizer, but are not (sufficiently) addressed in traditional parsing libraries:

1. We are only interested in sequences of transformation rules that can be applied successively to some initial term, and this is hard to express in most libraries. Parsing approaches that start by analyzing the grammar for constructing a parsing table will not work in our setting because they can not take the current term into account.
2. The ability to diagnose errors in the input highly influences the quality of the feedback. It is not enough to detect that the input is incorrect, but we also want to know at which point the input deviates from the strategy, and what is expected at this point. Some of the more advanced parser tools have error correcting facilities, which helps diagnosing an error to some extent.
3. Exercises are solved incrementally, and therefore we do not only have to recognize full sentences, but also prefixes. Backtracking and look-ahead can not be used because we want to recognize strategies at each intermediate step.
4. Labels help to describe the structure of a strategy in the same way as nonterminals do in a grammar. For a good diagnosis it is vital that a recognizer knows at each intermediate step where it is in the strategy.
5. A strategy should be serializable, for instance because we want to communicate with other e-learning tools and environments.

In earlier attempts to design a recognizer library for strategies, we tried to reuse an existing error-correcting parser combinator library [31], but failed because of the reasons listed above. The library we develop in this section is written in the functional programming language Haskell [28]. The code in this paper is almost complete and conforms to the Haskell 98 standard. Although the code is relatively short, we want to emphasize that the library has been tested in practice on different domains. For instance, strategies implemented for the domain of linear algebra are more complex than the strategy for logical expressions reported in this paper.

5.1. Representing grammars

Because strategies are grammars, we start by exploring a suitable representation for grammars. The data type for grammars is based on the alternatives of the strategy language discussed in Section 4:

$$\begin{aligned}\textbf{data}\ \mathit{Grammar}\ a = {} & \mathit{Grammar}\ a :\star: \mathit{Grammar}\ a \\ \mid {} & \mathit{Grammar}\ a :|: \mathit{Grammar}\ a \\ \mid {} & \mathit{Rec}\ \mathit{Int}\ (\mathit{Grammar}\ a) \\ \mid {} & \mathit{Symbol}\ a \mid \mathit{Var}\ \mathit{Int} \mid \mathit{Succeed} \mid \mathit{Fail}\end{aligned}$$

The type variable a in this definition is an abstraction for the type of the symbols: for strategies, the symbols are rules. The first design choice is how to represent

recursive grammars, for which we use the constructors *Rec* and *Var*. A *Rec* binds all the *Var*s in its scope that have the same integer. We assume that all our grammars are closed, i.e., there are no free occurrences of variables. This data type makes it easy to manipulate and analyze grammars. Alternative representations for recursion are higher-order fixed point functions, or nameless terms using de Bruijn indices. We discuss in Section 5.2 how *fix* is defined in terms of *Rec* and *Var*.

Labels are absent and will be added later. Observe that we use the constructors :⋆: and :|: for sequence and choice, respectively (instead of the combinators <⋆> and <|> introduced earlier). Haskell infix constructors have to start with a colon, but the real motivation is that we use <⋆> and <|> as smart constructors.

Although we use *Rec* and *Var* to represent recursive grammars, we use the fixed point representation instead to define recursive grammars. It is more convenient to use the fixed point representation to define recursion. The *fix* combinator is expressed as a smart constructor in terms of *Rec* and *Var*.

5.2. Smart constructors

A smart constructor is a normal function that in addition to constructing a value performs some checks or some simplifications. We use smart constructors for simplifying grammars, and to obtain a normal form. We introduce a smart constructor for every alternative of the *Grammar* data type: the functions *symbol*, *var*, *succeed*, and *fail* do nothing special, but are introduced for consistency.

The smart constructor <⋆> for sequences removes the unit element *Succeed*, and propagates the absorbing element *Fail*. Because the input is processed from left to right, we associate sequences to the right. Pay close attention to the occurrences of the smart constructors and the actual constructors in the following definition:

$$
\begin{array}{lll}
(<\star>) :: \mathit{Grammar}\ a \to \mathit{Grammar}\ a \to \mathit{Grammar}\ a & & \\
\mathit{Succeed} <\star> t & = t \\
s <\star> \mathit{Succeed} & = s \\
\mathit{Fail} <\star> \_ & = \mathit{fail} \\
\_ <\star> \mathit{Fail} & = \mathit{fail} \\
(s :\star: t) <\star> u & = s :\star: (t <\star> u) \\
s <\star> t & = s :\star: t
\end{array}
$$

For choices, we remove occurrences of *Fail*, and we nest alternatives to the right:

$$
\begin{array}{lll}
(<|>) :: \mathit{Grammar}\ a \to \mathit{Grammar}\ a \to \mathit{Grammar}\ a & & \\
\mathit{Fail} <|> t & = t \\
s <|> \mathit{Fail} & = s \\
(s :|: t) <|> u & = s :|: (t <|> u) \\
s <|> t & = s :|: t
\end{array}
$$

The smart constructor for recursive grammars checks that there is at least one free occurrence of the variable in the body: a *Rec* is built only if this is the case.

```
rec :: Int → Grammar a → Grammar a
rec i s = if i ∈ freeVars s then Rec i s else s
```

Calculating the set of free variables of a grammar is straightforward, although we have to take care of shadowing binders.

Finally, we define the constructor function for fixed points on grammars, which gives us another way to specify recursive grammars:

```
fix :: (Grammar a → Grammar a) → Grammar a
```

This function can be implemented using *rec* and *var*: the only difficulty in defining *fix* is to discover which integer can be used. We omit the implementation details.

5.3. Empty and firsts

For recognizing sentences, we have to define the functions *empty* and *firsts*. The function *empty* tests whether the empty sentence is part of the language.

```
empty :: Grammar a → Bool
empty (s :⋆: t)  = empty s ∧ empty t
empty (s :|: t)  = empty s ∨ empty t
empty (Rec i s)  = empty s
empty Succeed    = True
empty _          = False
```

The last definition covers the cases for *Fail*, *Symbol*, and *Var*. The most interesting definition is for the pattern (*Rec i s*): it calls *empty* recursively on *s* as there is no need to inspect recursive occurrences.

The function *firsts* returns a list with all symbols that can appear as the first symbol of a sentence. For each symbol, the function also returns the remaining grammar, i.e., the sentences that can appear after that symbol.

```
firsts :: Grammar a → [(a, Grammar a)]
firsts (s :⋆: t)  = [(a, s' <⋆> t) | (a, s') ← firsts s] ++
                    (if empty s then firsts t else [])
firsts (s :|: t)  = firsts s ++ firsts t
firsts (Rec i s)  = firsts (replaceVar i (Rec i s) s)
firsts (Symbol a) = [(a, succeed)]
firsts _          = []
```

For a sequence (*s* :⋆: *t*), we determine which symbols can appear first for *s*, and we change the results to reflect that *t* is part of the remaining grammar. Furthermore, if *s* can be empty, then we also have to look at the *firsts* for *t*. For choices, we simply combine the results for both operands. If the grammar is a single symbol,

then this symbol appears first, and the remaining strategy is *succeed* (we are done). To find the firsts for $(Rec\ i\ s)$, we have to look inside the body s. All occurrences of this recursion point are replaced by the grammar itself before we call *firsts* again. The replacement is performed by a helper-function: $replaceVar\ i\ s\ t$ replaces all free occurrences of $(Var\ i)$ in t by s.

The function *nonempty* removes the empty sentence from a grammar, and is defined using *firsts*:

$$\begin{aligned}&nonempty :: Grammar\ a \to Grammar\ a\\&nonempty\ s = foldr\ (<|>)\ fail\ [symbol\ a <\star> t \mid (a,t) \leftarrow firsts\ s]\end{aligned}$$

5.4. Running a strategy

So far, nothing specific about recognizing strategies has been discussed. A strategy is a grammar over rewrite rules: with the functions *empty* and *firsts* we can run a strategy with an initial term:

$$\begin{aligned}&run :: Grammar\ (Rule\ a) \to a \to [a]\\&run\ s\ a = [a \mid empty\ s] +\!\!+ [c \mid (r,t) \leftarrow firsts\ s, b \leftarrow apply\ r\ a, c \leftarrow run\ t\ b]\end{aligned}$$

This function is an implementation of the *run* function defined in Section 4.1. The list of results returned by *run* consists of two parts: the first part tests whether *empty* s holds, and if so, it yields the singleton list containing the term a. The second part takes care of the non-empty alternatives. Let r be one of the symbols that can appear first in strategy s (r is a rewrite rule). We are only interested in r if it can be applied to the current term a. It is irrelevant how the type *Rule* is defined, except that applying a rule to a term returns a list of results. We run the remainder of the strategy (that is, t) with the result of the application of rule r.

The function *run* can produce an infinite list. In most cases, however, we are only interested in a single result (and rely on lazy evaluation). The part that considers the empty sentence is put at the front to return sentences with few rewrite rules early. Nonetheless, the definition returns results in a depth-first manner. We define a variant of *run* which exposes breadth-first behavior:

$$\begin{aligned}&run' :: Grammar\ (Rule\ a) \to a \to [[a]]\\&run'\ s\ a = [a \mid empty\ s] : merge\ [run'\ t\ b \mid (r,t) \leftarrow firsts\ s, b \leftarrow apply\ r\ a]\\&\quad \textbf{where}\ merge = map\ concat \circ transpose\end{aligned}$$

The function run' produces a list of lists: results are grouped by the number of rewrite steps that have been applied, thus making explicit the breadth-first nature of the function. The helper-function *merge* merges the results of the recursive calls: by transposing the list of results, we combine results with the same number of steps.

5.5. Labels

Labels are not included in the *Grammar* data type. We introduce two mutually recursive types for strategies that can have labeled parts:

```
data LabeledStrategy l a = Label l (Strategy l a)
type Strategy        l a = Grammar (Either (Rule a) (LabeledStrategy l a))
```

A labeled strategy is a strategy with a label (of type l). A strategy is a grammar where the symbols are either rules or labeled strategies. For this choice, we use the *Either* data type: rules are tagged with the *Left* constructor, labeled strategies are tagged with *Right*. With the type definitions above, we can have grammars over other grammars, and the nesting can be arbitrarily deep.

Excluding labels from the *Grammar* data type is a design choice. Functions that work on the *Grammar* data type don't have to deal with labels, which makes it, for example, simpler to manipulate grammars. A disadvantage of our solution is that symbols in a strategy must be tagged *Left* or *Right*. In our actual implementation we circumvent the tagging by overloading the strategy combinators. As a result, strategies can really be defined as the specifications in Section 4.

Now that we can label parts of a strategy, we want to keep track at which point in the strategy we are, and we do so without changing the underlying machinery. We start with defining the *Step* helper data type:

```
data Step l a = Enter l | Step (Rule a) | Exit l
```

A step is a rewrite rule (constructor *Step*), or a constructor to indicate that we entered (or left) a labeled part of the strategy. A labeled strategy can be turned into a grammar over steps in the following way:

```
withSteps :: LabeledStrategy l a → Grammar (Step l a)
withSteps (Label l s) = symbol (Enter l)
                      <*> mapSymbol (either (symbol ∘ Step) withSteps) s
                      <*> symbol (Exit l)
```

For each label, we introduce symbols that mark the beginning and the end of that label. We use the function *mapSymbol* to transform strategy s to a grammar of steps. The function $mapSymbol :: (a \rightarrow Grammar\ b) \rightarrow Grammar\ a \rightarrow Grammar\ b$ applies its argument function f to all symbols in a grammar. Note that f returns a grammar, and therefore can be used to change symbols and to flatten a grammar of grammars in one traversal. Each symbol of s is either a rewrite rule or a labeled strategy (see the type definition of *Strategy*): a rewrite rule becomes a symbol with a step, and a labeled strategy is handled by calling the function *withSteps* recursively.

To run a grammar with steps, we first have to overload the function *apply* such that it also works on *Step*, and generalize the types of *run* and run' accordingly. The step data type gives us more information, as we show in our next example.

A derivation. Suppose that we run *dnfStrategy2* from Section 4 on the term $\neg(\neg(p \vee q) \wedge r) \wedge T$: what would be the result? Of course, we would expect to get the derivation:

$$
\begin{array}{ll}
 & \neg(\neg(p \vee q) \wedge r) \wedge T \\
\equiv & \quad \{\textsc{AndTrue}\} \\
 & \neg(\neg(p \vee q) \wedge r) \\
\equiv & \quad \{\textsc{DeMorganAnd}\} \\
 & \neg\neg(p \vee q) \vee \neg r \\
\equiv & \quad \{\textsc{NotNot}\} \\
 & p \vee q \vee \neg r
\end{array}
$$

The final answer, $p \vee q \vee \neg r$, is indeed what we get. It is informative to step through the above derivation and see the intermediate steps.

[*Enter* ℓ_1,	*Step* AndTrue,	*Step not*,	*Exit* ℓ_1,
Enter ℓ_2,	*Step not*,	*Exit* ℓ_2,	*Enter* ℓ_3,
Step DeMorganAnd,	*Step not*,	*Step down*,	*Step* NotNot,
Step up,	*Step not*,	*Exit* ℓ_3,	*Enter* ℓ_4,
Step not,	*Exit* ℓ_4]		

The list has many steps, but only three correspond to actual steps from the derivation: the rules of those steps are underlined. The other rules are administrative: the rules *up* and *down* are introduced by the *somewhere*, *bottomUp*, and *topDown* combinators, whereas *not* comes from the use of *repeat*. Also observe that each *Enter* step has a matching *Exit* step. In principle, a label can be visited multiple times by a strategy.

5.6. Parallel strategies

The parallel combinator is added as a primitive to our *Grammar* data type. As explained in Section 4.6, the parallel combinator does not have to be defined as a primitive, but by not doing so, some strategies become infeasibly large. In our setup, it is relatively easy to add a new constructor to the *Grammar* data type.

```
data Grammar a = ... | Grammar a :||: Grammar a
```

Just as we did for sequences and choices, we first introduce a smart constructor for parallel strategies, which expresses that it has *Succeed* as its unit element, *Fail* as its absorbing element, and that the combinator is associative:

```
(<||>) :: Grammar a → Grammar a → Grammar a
Succeed <||> t       = t
s       <||> Succeed = s
Fail    <||> _       = fail
_       <||> Fail    = fail
```

$$
\begin{aligned}
(s \mathbin{:||:} t) \mathbin{<||>} u &= s \mathbin{:||:} (t \mathbin{<||>} u) \\
s \mathbin{<||>} t &= s \mathbin{:||:} t
\end{aligned}
$$

Next, we extend the definitions of *empty* and *firsts* with a new case:

$$
\begin{aligned}
\mathit{empty}\ (s \mathbin{:||:} t) &= \mathit{empty}\ s \wedge \mathit{empty}\ t \\
\mathit{firsts}\ \ (s \mathbin{:||:} t) &= [(a, s' \mathbin{<||>} t\ ) \mid (a, s') \leftarrow \mathit{firsts}\ s] +\!\!+ \\
&\quad\ [(a, s\ \ \mathbin{<||>} t') \mid (a, t') \leftarrow \mathit{firsts}\ t]
\end{aligned}
$$

Other functions that operate on the *Grammar* data type (such as *freeVars* and *mapSymbol*) have to be extended as well, but these changes are minimal. Using a generic traversal library [23] can further reduce the impact of adding a constructor.

In a similar way, we can define useful variants on this combinator, such as a left-biased parallel combinator (which continues with its left operand strategy whenever this is possible), or a parallel combinator that stops as soon as one of its operand strategies is finished.

6. Feedback on strategies

This section briefly sketches how we use the strategy language, as introduced in the previous sections, to give feedback to users of our e-learning systems, or to users of other e-learning systems that make use of our feedback services [14]. We have implemented several kinds of feedback. Most of these categories of feedback appear in the tutoring principles of Anderson [1], or in existing tools supporting the stepwise construction of a solution to an exercise. No existing tool implements more than just a few of these categories of feedback.

We do not try to tackle the problem of how feedback should be presented to a student in this paper. Here we look at the first step needed to provide feedback, namely to diagnose the problem, and relate the problem to the rules and the strategy for the exercise. We want users of our feedback services to determine how these findings are presented to the user. For example, we could generate a table from the strategy with the possible problems, and let teachers fill this table with the desired feedback messages.

Feedback after a step. After each step performed by a student, we check whether or not this step is valid according to the strategy. For steps involving argument- and variable-value computations we have to resort to generators, which calculate the correct values of these components, and check these values against the values supplied by the student. These generators are easily and naturally expressed in our framework.

Checking whether or not a step is valid amounts to checking whether or not the sequence of steps supplied by the student is a valid prefix of a sentence of the language specified by the context-free grammar corresponding to the strategy. As

soon as we detect that a student no longer follows the strategy, we have several possibilities to react. We can force the student to undo the last step, and let the student strictly follow the strategy. Alternatively, we can warn the student that she has made a step that is invalid according to the strategy, but let the student proceed on her own path. For instance, a student has to rewrite $\neg(\neg(p \vee q) \wedge r) \wedge T$ to DNF using *dnfStrategy2*. When applying the DeMorganAnd rule, the student can be warned that although the step is correct, it is better to do something else (removing the constants).

Another example is when a student submits $\neg\neg(p \vee q) \vee \neg r$ as the final answer. The exercise assistant reports back to the student that his answer is equivalent, but with a gentle reminder that the exercise is not yet finished. In this scenario, the strategy tells exactly which step needs to be taken: the double negation should be simplified.

Progress. Given an exercise and a strategy for solving the exercise, we can determine the minimum number of steps necessary to solve the exercise, and show this information in a progress bar. Each time a student performs a correct step, the progress bar is updated.

Strategy unfolding. We have constructed an OpenMath binding with the MathDox system [11], in which a student enters a final answer to a question. If the answer is incorrect, we return a new exercise, which is part of the initial exercise. For example, if a student does not return a correct disjunctive normal form of a logical expression, we ask the student to solve the simpler problem of first eliminating all occurrences of T and F in the logical expression. After completing this part of the exercise, we ask to solve the remaining part of the exercise. This kind of feedback is also used by Cohen et al. [10] in an exercise assistant for calculus.

Hint. A student can ask for a hint. Given an exercise and a strategy for solving the exercise, we calculate the 'best' next step. The best next step is an element of the first set of the context-free grammar specified by the strategy. For example, a student has no clue how to rewrite $\neg(\neg(p \vee q) \wedge r) \wedge T$ to DNF, and presses a hint button. The system reports the hint: "remove all constants". The fact that the constants have to be removed can be concluded from the strategy. If this does not help the student, the system can emit a more specific message that suggests a specific rewrite rule (or the result of applying that rule).

An alternative possibility for hints is to use strategy unfolding. Instead of giving the 'best' next step when asked for a hint, we tell the student which sub-problem should be solved first.

Completion problems. Worked-out problems can be generated from a strategy, showing all the steps from the initial term to the expected answer. A worked-out problem is a presentation of a sentence that is generated by the strategy, like the derivation shown in Subsection 5.5. Sweller, based on his cognitive load theory,

describes a series of effects and guidelines to create learning materials. The basic idea is that a student profits from having example solutions played for him or her, followed by an exercise in which the student fills out some missing steps in a solution [30]. We can use the strategy for a problem to play a solution for a student, and we can play all but the middle two (three, last two, etc.) steps, and ask the student to complete the exercise.

Buggy rules. Strategies can refer to buggy rules: these rules capture common mistakes, and make it possible to report specialized messages for often occurring errors. For example, a student submits $\neg\neg(p \vee q) \wedge \neg r$ as an intermediate solution to the exercise $\neg(\neg(p \vee q) \wedge r)$. Because the terms are not equivalent, the buggy rules are considered (BuggyDM1 and BuggyDM2), and in this case, rule BuggyDM1 matches. A special message associated with this rule (for example, "when applying the DeMorganAnd rule, the $\wedge$ operator must be replaced by the $\vee$ operator") is reported to the student.

Buggy strategies. If a step supplied by a student is invalid with respect to the strategy specified, but can be explained by a buggy strategy for the problem, we give the error message belonging to the buggy strategy. Again, this amounts to parsing, not just with respect to the correct strategy, but also with respect to known buggy strategies. In our logic domain, for example, a buggy strategy would be to use the distribution rule before pushing the negations inside the expression.

7. Conclusions

We have introduced a strategy language with which we can specify strategies for exercises in many domains. A strategy is defined as a context-free grammar, extended with non-context-free constructs for, for example, manipulating variables and arguments. The formulation of a strategy as a context-free grammar allows us to automatically calculate several kinds of feedback to students incrementally solving exercises. Languages for specifying procedures or strategies for exercises have been developed before. Our language has the same expressive power and structure; our main contribution is the advanced feedback we can calculate automatically, and relatively easily. This is achieved by separating the strategy language into a context-free language, the strategy combinators, and a non-context-free language, the embedding as a domain-specific language. In Section 6 we have shown how strategies can be used to report advanced feedback.

We have presented a complete implementation of a recognizer for strategies. Although it is tempting to reuse existing parsing tools and libraries, a closer look at the problem reveals subtle differences that make the existing tools unsuitable for dealing with the problem we are facing. Some design choices were discussed, in particular how to deal with recursion, and how to mark positions in a strategy.

We have several plans for the future. We hope to create bindings of our feedback service with more existing tools, such as ActiveMath [15]. For this purpose, we need to standardize the protocol for providing feedback. Our tool already has a binding with MathDox [11], and has recently been used in a classroom setting. We have collected data from these sessions and preliminary analyses show that providing feedback on the strategic level improves far transfer: students that received feedback on the strategic level did better in advanced exercises. We will collect more data from the experiments, and analyze and report on the results. Also, we want to apply our ideas to domains with less structure, such as computer programming [3], software modeling, and maybe even serious games in which students have to cooperate to achieve a certain goal. A final area that requires further investigation is how to make strategies more accessible to teachers.

Acknowledgments. This work was made possible by the support of the SURF Foundation, the higher education and research partnership organization for Information and Communications Technology (ICT). For more information about SURF, please visit `http://www.surf.nl`. We thank the anonymous reviewers for their constructive comments. Discussions with Hans Cuypers, Josje Lodder, Wouter Pasman, Rick van der Meiden, Erik Jansen, and Arthur van Leeuwen are gratefully acknowledged.

References

[1] John R. Anderson. *Rules of the Mind*. Lawrence Erlbaum Associates, 1993.

[2] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. Cognitive tutors: lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995.

[3] John R. Anderson and Edward Skwarecki. The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9):842–849, 1986.

[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] Eric Bouwers. Improving automated feedback – a generic rule-feedback generator. Master's thesis, Utrecht University, department of Information and Computing Sciences, 2007.

[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`, May 2008.

[10] Arjeh Cohen, Hans Cuypers, Dorina Jibetean, and Mark Spanbroek. Interactive learning and mathematical calculus. In *Mathematical Knowledge Management*, 2005.

[11] Arjeh Cohen, Hans Cuypers, Ernesto Reinaldo Barreiro, and Hans Sterk. Interactive mathematical documents on the web. In *Algebra, Geometry and Software Systems*, pages 289–306. Springer-Verlag, 2003.

[12] Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. *Electron. Notes Theor. Comput. Sci.*, 174(1):17–34, 2007.

[13] Ido Erev, Adi Luria, and Anan Erev. On the effect of immediate feedback, 2006. `http://telem-pub.openu.ac.il/users/chais/2006/05/pdf/e-chais-erev.pdf`.

[14] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Sylvia Stuurman. Feedback Services for Exercise Assistants. In Dan Remenyi, editor, *The Proceedings of the 7th European Conference on e-Learning*, pages 402–410. Academic Publishing Limited, 2008.

[15] G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. In *International Conference on Computers in Education, ICCE 2005*, 2005.

[16] Bastiaan Heeren, Johan Jeuring, Arthur van Leeuwen, and Alex Gerdes. Specifying strategies for exercises. Technical Report UU-CS-2008-001, Utrecht University, 2008.

[17] 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.

[18] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 72–149. Springer-Verlag, 2007.

[19] Helmut Horacek and Magdalena Wolska. Handling errors in mathematical formulas. In M. Ikeda, K. Ashley, and T.-W. Chan, editors, *ITS 2006*, volume 4053 of *LNCS*, pages 339–348. Springer-Verlag, 2006.

[20] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.

[21] Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[22] Marina Issakova. *Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment*. PhD thesis, University of Tartu, 2007.

[23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. TLDI'03.

[24] Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. 18 p.; Draft; Available at `http://www.cwi.nl/~ralf`, October15 2002.

[25] 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.

[26] 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.

[27] L.C. Paulson. *ML for the Working Programmer, 2nd Edition*. Cambridge University Press, 1996.

[28] 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/`.

[29] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *ICFP '07*, pages 141–152, New York, NY, USA, 2007.

[30] J. Sweller, J.J.G. van Merriënboer, and F. Paas. Cognitive architecture and instructional design. *Educational Psychology Review*, 10:251–295, 1998.

[31] 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*, pages 184–207. Springer-Verlag, 1996.

[32] Kurt VanLehn. *Mind Bugs – The Origins of Procedural Misconceptions*. MIT Press, 1990.

[33] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98*, pages 13–26, 1998.

[34] Claus Zinn. Supporting tutorial feedback to student help requests and errors in symbolic differentiation. In M. Ikeda, K. Ashley, and T.-W. Chan, editors, *ITS 2006*, volume 4053 of *LNCS*, pages 349–359. Springer-Verlag, 2006.

Bastiaan Heeren
School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
`bhr@ou.nl`

Johan Jeuring
School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
`jje@ou.nl`
Department of Information and Computing Sciences, Universiteit Utrecht

Alex Gerdes
School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
`age@ou.nl`