Math.Comput.Sci. (2010) 3:349–370
DOI 10.1007/s11786-010-0027-4

**Mathematics in Computer Science**

# Specifying Rewrite Strategies for Interactive Exercises

**Bastiaan Heeren · Johan Jeuring · Alex Gerdes**

Received: 26 January 2009 / Revised: 30 July 2009 / Accepted: 10 December 2009 / Published online: 10 March 2010


**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 examples, 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 rewrite rules, which turns tracking intermediate steps into a parsing problem. This is a promising view at interactive exercises 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, we discuss generating feedback on strategies, and the implementation of a strategy recognizer.

**Keywords** Strategy language · Interactive exercises · Feedback · Recognizer · Context-free grammar

## 1 Introduction

Tools like Aplusix [12], Active-Math [21], MathPert [4], the Freudenthal Digital Mathematics Environment (DWO) [5,18], and our own tool for rewriting logic expressions [32] 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:

B. Heeren · J. Jeuring · A. Gerdes
School of Computer Science, Open Universiteit Nederland, P. O. Box 2960, 6401 DL Heerlen, The Netherlands
e-mail: bhr@ou.nl

A. Gerdes
e-mail: age@ou.nl

J. Jeuring (✉)
Department of Information and Computing Sciences, Universiteit Utrecht, P. O. Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail: johanj@cs.uu.nl; jje@ou.nl

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 [39]. The second kind of error is a rewriting error: the student rewrites an expression using a non-existing or buggy rule [7,23]. 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 an embedded domain-specific language (EDSL), in which strategies are specified as context-free grammars (CFGs). 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 (see Sect. 3). Then, we present our language for writing strategies in Sect. 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. In Sect. 5 we discuss several kinds of feedback and hints made possible by our strategy language. The design and implementation of a strategy recognizer is briefly discussed in Sect. 6. The last two sections present related work, conclusions, and 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 [9]:

- 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 [9], procedural nets [7], plans, tactics, etc.), and we will use this term in the rest of the 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.

### 2.1 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 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 (see [40][1] for a list of almost 50 tools for teaching logic), physics, etc. have been developed.

### *2.1.1 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. [17] also claim that immediate feedback is often to be preferred.

### *2.1.2 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 detailed feedback at intermediate steps based on student input. The DWO [5] labels intermediate steps with a green (correct) or red (wrong) symbol. MathDox [14] can provide more detailed feedback, which has to be specified together with the exercise, leading to exercise files of hundreds of lines. Active-Math [21] gives feedback for classes of exercises, but doesn't take student input into account when providing feedback.

Although all these kinds of feedback at intermediate steps are 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 [23] on student bugs in calculating fractions), and automatically calculating feedback for a given exercise, strategy, and student input is very difficult.

### *2.1.3 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 Sect. 4.6.

[1] These pages contain a comprehensive, alphabetically ordered list of educational logic software.

### 2.2 Representing Strategies

Representing a 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.

### 2.3 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 EDSL in a programming language [26] 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.

### 2.4 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. However, we do intend to include facilities for building a student model in the future, to offer the possibility to select tasks that are suitable for a student.

### 2.5 How do we Calculate Feedback on Strategies?

The strategy language is defined as an EDSL in Haskell [29]. Using the basic constructs from the strategy language, we can create something that resembles a CFG. The sentences of this grammar are sequences of rewrite steps (applications of rules). We can thus check whether or not a student follows a strategy by parsing the sequence of rewrite steps, and checking that the sequence of rewrite steps is a prefix of a correct sentence from the CFG. 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 CFG, 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.

## 3 Three Example Strategies

In this section we present three strategies for rewriting a classical logical expression to DNF. 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:

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

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 DNF.

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.

### 3.1 Strategy 1: Apply Rules Exhaustively

The first strategy applies the basic rules from Fig. 1 exhaustively: we proceed as long as we can apply *some* rule *somewhere*, and we will end up with a logical expression in DNF. 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.

### 3.2 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.

*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: | $p \leftrightarrow \neg p$ | $= F$ |

**Fig. 1** Rules for logical expressions

- **Step 1:** Remove constants from the logical expression with the rules for "constants" (see Fig. 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.

### 3.3 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 Fig. 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:

BuggyDM1 : $\neg(p \wedge q) \neq \neg p \wedge \neg q$ 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 CFGs. In fact, we can view a strategy as a grammar where the rules form the alphabet of the language.

- **Rewrite 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 rewrite rule $r$ to distinguish the strategy from the rewrite rule.
- **Sequence.** Two strategies can be composed and put in sequence. We write $s$ <⋆> $t$ to denote the sequence of strategy $s$ followed by strategy $t$.
- **Choice.** We can choose between two strategies, for which we will write $s$ <|> $t$. 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 <⋆> 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* $\ell$ $s$ to label strategy $s$ with some label $\ell$. 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 rewrite rules.

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

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:

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

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 Fig. 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:

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

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:

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

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

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

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 $a$:

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

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.

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

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:

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

Using generic programming techniques [24], 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).

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

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

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

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 Sect. 3, we presented three alternative strategies for turning a logical expression into DNF. 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 Fig. 1:

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

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

$$
\begin{aligned}
dnfStrategy1 &= repeat\ (somewhere\ basicRules) \\
dnfStrategy2 &= label\ \texttt{"Constants"}\ (repeat\ (topDown\ constants)) \\
&\quad <\star>\ label\ \texttt{"Definitions"}\ (repeat\ (bottomUp\ definitions)) \\
&\quad <\star>\ label\ \texttt{"Negations"}\ (repeat\ (topDown\ negations)) \\
&\quad <\star>\ label\ \texttt{"Distribution"}\ (repeat\ (somewhere\ distribution))
\end{aligned}
$$

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 Sect. 4.7.

## 4.6 Gaussian Elimination Strategy

We present a strategy from the domain of linear algebra to illustrate the strategy language. This strategy brings a matrix into reduced echelon form, following a procedure known as Gaussian elimination. This example is more involved than the strategies for DNF, and shows that additional information can be maintained in a context while running a strategy.

We start by giving an informal description of the Gaussian elimination procedure. In this procedure we use two variables: the $j$-th column of the matrix, and the number of rows covered. The procedure consists of a forward pass, followed by a backward pass.

**Forward pass:**

(a) let $j$ be the first column with a non-zero entry
(b) exchange the first row (if necessary) to get a non-zero entry in $j$, and scale the row (if necessary) to get a one
(c) add the first row to the other rows to get zeros in $j$
(d) cover up the first row, and repeat steps *(a)* to *(c)*

**Backward pass:**

(e) ...

Gaussian elimination revolves around three elementary operations on rows: interchanging two rows, scaling a row, and adding a row a specified number of times to another row. These operations are visible in the following worked-out example (one of the derivations that follow the procedure):

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 2 & 3 & 2 \\ 3 & 1 & 1 & 3 \end{pmatrix} \overset{exchange}{\Longrightarrow} \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 1 & 1 & 1 \\ 3 & 1 & 1 & 3 \end{pmatrix} \overset{add}{\Longrightarrow} \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & -5 & -8 & -3 \end{pmatrix}$$

$$\overset{add}{\Longrightarrow} \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & -3 & 2 \end{pmatrix} \overset{scale}{\Longrightarrow} \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & -\frac{2}{3} \end{pmatrix} \overset{add}{\Longrightarrow} \begin{pmatrix} 1 & 2 & 0 & 4 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & -\frac{2}{3} \end{pmatrix}$$

$$\overset{add}{\Longrightarrow} \begin{pmatrix} 1 & 2 & 0 & 4 \\ 0 & 1 & 0 & \frac{5}{3} \\ 0 & 0 & 1 & -\frac{2}{3} \end{pmatrix} \overset{add}{\Longrightarrow} \begin{pmatrix} 1 & 0 & 0 & \frac{2}{3} \\ 0 & 1 & 0 & \frac{5}{3} \\ 0 & 0 & 1 & -\frac{2}{3} \end{pmatrix}$$

We now turn our attention to specifying the procedure in our strategy language. The procedure consists of two passes, and we reflect this in our strategy specification by composing the two passes: these passes can now be reused as part of other strategies.

```
toReducedEchelon = label "Gaussian elimination"
                         (forwardPass <*> backwardPass)
```

The forward pass is a strategy consisting of five parts, and these parts have to be repeated until this is no longer possible (see step *d*). Note that step *b* consists of two parts, and both are optional ("if necessary"). This leads to the following strategy specification:

```
forwardPass = label "Forward pass" (
  repeat   (  label "Find j-th column"      FindColumnJ
         <*> label "Exchange rows"          (try ExchangeNonZero)
         <*> label "Scale row"              (try ScaleToOne)
         <*> label "Zeros in j-th column"   (repeat ZerosFP)
         <*> label "Cover up top row"       CoverRow
          ))
```

The labels in this specification correspond to the hierarchical structure of the procedure. Remember that labels help to tailor feedback generation.

Let us have a closer look at the rules FindColumnJ and CoverRow. These rewrite rules are so-called *administrative rules* because they only change the context containing the variables *column-j* and *rows-covered*, not the matrix. In fact, steps *a* and *d* of the informal description did not result in steps in the worked-out example. The other three rewrite rules in the forward pass (ExchangeNonZero, ScaleToOne, and ZerosFP) correspond directly to the three elementary row operations. These rules use the variables *column-j* and *rows-covered* to determine how the row operations are performed.

The structure of the backward pass is quite similar to the forward pass. In the following definition, UncoverRow is again an administrative rule:

```
backwardPass = label "Backward pass" (
  repeat   (  label "Uncover row" UncoverRow
         <*> label "Sweep"        (repeat ZerosBP)
          ))
```

### 4.7 Parallel Strategies

Suppose that we want to run the strategies *s* and *t* in parallel, denoted by $s <||> t$. 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.

```
dnfStrategy3 = whenever ((tautologies <|> contradictions) <*> step1)
                        dnfStrategy2
```

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

### 4.8 Reflections

Figure 2 presents an overview of the strategy combinators introduced so far. Is this 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

| Combinator | Description |
|---|---|
| $s$ <*> $t$ | first $s$, then $t$ |
| $s$ <\|> $t$ | either $s$ or $t$ |
| *succeed* | ever succeeding strategy |
| *fail* | ever failing strategy |
| *fix f* | fixed point combinator |
| *label* $\ell$ $s$ | attach label $\ell$ to $s$ |
| *many* $s$ | apply $s$ zero or more times |
| *many1* $s$ | apply $s$ one or more times |
| *option* $s$ | either apply $s$ or not |

| Combinator | Description |
|---|---|
| *not* $s$ | succeeds if strategy $s$ is not applicable |
| *repeat* $s$ | apply $s$ as long as possible |
| *repeat1* $s$ | as *repeat*, but at least once |
| *try* $s$ | apply $s$ once if possible |
| $s \triangleright t$ | apply $s$, or else $t$ |
| *somewhere* $s$ | apply $s$ at some location |
| *bottomUp* $s$ | search location bottom-up |
| *topDown* $s$ | search location top-down |

**Fig. 2** Summary table of strategy combinators

strategic programming languages such as Stratego [31,43], and very similar languages are used in parser combinator libraries [27,39], boiler-plate libraries [30], workflow applications [37], theorem proving (tacticals [36]) and data-conversion libraries [16], which suggests that our library could serve as a firm basis for strategy specifications.

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 consist of the Gaussian elimination procedure (or one of its two passes), 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). If there are several different but acceptable approaches to solving an exercise, these can be combined into a single strategy by means of the choice-combinator. 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. It is advisable to ensure that clever shortcuts are accepted, by either adding them to the strategy, or by allowing deviations from the strategy, otherwise students are likely to get frustrated. In more complex domains, such as theorem proving, it is often impossible to predict all ways in which an exercise can be solved. Strategies in such domains are incomplete in that they don't describe all ways in which an exercise can be solved.

Our strategy language is particularly suited for describing procedures for solving exercises in well-structured domains. In such domains it is easy to formally describe the rewrite rules that can be applied to expressions within the domain. It is much harder to use strategies in domains with less structure. For example, we do not know how to write a strategy for constructing a UML-model for a piece of software, giving a textual description of the purposes of the software.

Our strategies should not be confused with tutorial strategies, which describe how to teach particular content. For example, a tutorial strategy may describe that it is better to first give a worked-out example, and only then require students to take some intermediate steps themselves, as in the 4C/ID-model [41]. Or it may describe that before giving exercises, the complete procedure should be explained and the underlying theorems be proved. We think our strategies can be used in any of such tutorial strategies.

## 5 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 [20]. 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.

We discuss a number of possible ways in which our strategies can be used for generating feedback automatically. This discussion is quite informal: in Sect. 6.7 we explain how these feedback categories relate to the implementation of the strategy recognizer.

### 5.1 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 calculate the correct values of these components, and check these values against the values supplied by the student. Such calculations 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 CFG 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 (remove the constant).

### 5.2 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 CFG 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 calculated 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).

### 5.3 Strategy Unfolding

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 subproblem should be solved first. The decomposition of a strategy into subproblems is based on the labels that are attached to the strategy. Such a subproblem can be decomposed too, until we reach a single rule.

We have constructed an OpenMath binding with the MathDox system [14], 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 makes a mistake in bringing a matrix into reduced echelon form, we ask the student to solve the subproblem of performing the forward pass (see the strategy definition in Sect. 4.6). 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. [13] in an exercise assistant for calculus.

### 5.4 Ready

After having performed some steps, a student can indicate to have finished the exercise at hand. For example, a student submits $\neg\neg(p \vee q) \vee \neg r$ as the final answer. Based on the strategy, the exercise assistant reports back to the student that the exercise is not yet finished (even though the expression is in DNF). In this scenario, the strategy tells exactly which step needs to be taken: the double negation should be simplified.

### 5.5 Completion Problems

Worked-out examples can be generated from a strategy, showing all the steps from the initial term to the expected answer. A worked-out example is a presentation of a sentence that is generated by the strategy. For example, the following derivation is generated by *dnfStrategy2* from Sect. 4:

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

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 [38,41]. 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.

### 5.6 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.

### 5.7 Buggy Rules

Buggy 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.

### 5.8 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.

## 6 Implementation of a Strategy Recognizer

In this section we discuss the design and implementation of a strategy recognizer. With this implementation we can precisely describe how the different types of feedback (listed in Sect. 5) can be realized. We highlight the most important parts and some interesting design choices: the implementation details are presented in a different paper [22]. This section is somewhat technical, and can be safely skipped by the reader not interested in the underlying implementation.

Recognizing a strategy comes down to tracing the steps that a student is taking. 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, and in the remainder we assume to know which rules have been applied. 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.

### 6.1 Design Considerations

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 [27,39], 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 rewrite 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 cannot 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 help 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 non-terminals 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 [39], but failed because of the reasons listed above. The code for strategy recognition is written in the functional programming language Haskell [29], just as the strategy combinators. Although the code is relatively short, we want to emphasize that the library has been tested in practice on different domains. The code fragments are executable, even though they may appear to be specifications rather than implementations.

### 6.2 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 Sect. 4:

```
data Grammar a = Grammar a  :⋆: Grammar a      -- sequence
               | Grammar a :|: Grammar a       -- choice
               | Symbol a                      -- symbol
               | Rec Int (Grammar a) | Var Int -- recursion
               | Succeed | Fail                -- units
```

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. This data type makes it easy to manipulate and analyze grammars. 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 can be 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. These smart constructors have the same type signature as the data type constructors:

```
(<⋆>), (<|>) :: Grammar a → Grammar a → Grammar a
```

In addition to combining two grammars, these constructors simplify the grammar, for instance by considering unit elements and absorbing elements. Likewise, we introduce the smart constructors *symbol*, *succeed*, and *fail*.

### 6.3 Empty and Firsts

For recognizing sentences, it suffices 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 [])
```

$$
\begin{aligned}
&\mathit{firsts}\ (s :|: t) &&= \mathit{firsts}\ s +\!\!+ \mathit{firsts}\ t \\
&\mathit{firsts}\ (\mathit{Symbol}\ a) &&= [\,(a, \mathit{succeed})\,] \\
&\mathit{firsts}\ (\mathit{Rec}\ i\ s) &&= \mathit{firsts}\ (\mathit{replaceVar}\ i\ (\mathit{Rec}\ i\ s)\ s) \\
&\mathit{firsts}\ \_ &&= [\,]
\end{aligned}
$$

For a sequence $(s :\star: t)$, 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 $(\mathit{Rec}\ i\ s)$, 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 $(\mathit{Var}\ i)$ in $t$ by $s$.

## 6.4 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. We use Haskell's list comprehensions for a concise implementation of the function *run*:

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

This function is an implementation of the *run* function defined in Sect. 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. A variant of this function that exposes breadth-first behavior can be defined straightforwardly.

The definition given for *run* only returns final terms. The function can of course be adapted such that it also returns all intermediate terms for a sentence, and all the rules that have been applied.

## 6.5 Labels

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{LabeledStrategy}\ l\ a = \mathit{Label}\ l\ (\mathit{Strategy}\ l\ a) \\
&\textbf{type}\ \mathit{Strategy}\ \ \ l\ a = \mathit{Grammar}\ (\mathit{Either}\ (\mathit{Rule}\ a)\ (\mathit{LabeledStrategy}\ l\ a))
\end{aligned}
$$

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 Sect. 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 with the following function (we omit its implementation):

*withSteps* :: *LabeledStrategy l a* → *Grammar* (*Step l a*)

We can now run a labeled strategy and inspect the intermediate steps. The step data type provides us with a lot of information, as we show in the next section.

### 6.6 Inspecting Intermediate Steps

We return to the derivation presented in Sect. 5.5. Running *dnfStrategy2* on the term $\neg(\neg(p \vee q) \wedge r) \wedge T$ results in the following derivation:

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

It is informative to step through the above derivation and see the intermediate steps (using *withSteps*). The labels in the list originate from *dnfStrategy2*. For instance, label `"Constants"` corresponds to the application of the constant rules.

```
[Enter "Constants",      Step AndTrue,          Step not,
Exit "Constants",        Enter "Definitions",   Step not,
Exit "Definitions",      Enter "Negations",     Step DeMorganAnd,
Step not,                Step down,             Step NotNot,
Step up,                 Step not,              Exit "Negations",
Enter "Distribution",    Step not,              Exit "Distribution"]
```

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.

### 6.7 Implementing Feedback on Strategies

We conclude the discussion on the strategy recognizer by revisiting the feedback categories listed in Sect. 5. These categories are closely related to the functions developed for the recognizer.

- For both feedback after a step and hint generation (Sects. 5.1 and 5.2) we can use *firsts*. For the former we check that the term submitted by the student is part of the list generated by *firsts*, for the latter we use the list to supply a hint. The function *empty* is used to check whether we have finished the exercise or not (Sect. 5.4).
- Strategy unfolding (Sect. 5.3) is remarkably easy with the information that becomes available with the *Step* data type and the *withSteps* function. The intermediate steps tell us exactly where we are in the strategy. Our data type representation for strategies makes it possible to decompose a strategy into substrategies.
- The function for running a strategy is the basis for generating and checking completion problems (Sect. 5.5): this function returns all the information needed for a worked-out example. By omitting steps, we turn the worked-out example into a completion problem. By taking the length of the worked-out example, we can report progress information (Sect. 5.6), i.e., the number of remaining steps.
- All we need for common misconceptions (buggy rules and buggy strategies, Sects. 5.7 and 5.8) is the ability to label part of a strategy as buggy. If a label corresponding to a buggy strategy (or buggy rule) shows up in the list of intermediate steps, we can react accordingly.

## 7 Related Work

Using rewrite strategies for specifying exercises, and using parsers to automatically calculate different kinds of feedback is a novel idea, as far as we are aware. However, providing feedback to students about several aspects of their exercises has been and still is an active area of research.

Explaining syntax errors has been studied in several contexts, most notably in compiler construction [39], but also for e-learning tools [25]. Some work has been done on trying to explain errors made by students on the level of rewrite rules [6,23,28,35].

Already around 1980, but also later, Brown and Burton [7], Brown and VanLehn [8] and VanLehn [42], 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 [42], 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 similar to a language for CFGs for specifying 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.

Tacticals [36] and proof plans and methods [10] are used to automatically prove theorems. On an abstract level, these plans and methods play the same role as strategies: we can view a strategy as a proof plan for proving that an answer is the solution to an exercise. Bundy [11] discusses how proof plans are used to support interactive proving, by letting the user help the theorem prover whenever the theorem prover cannot make progress anymore, or by letting the prover explain why a particular method can or cannot be applied. As far as we found, proof plans are not used to teach theorem proving, or to recognize proving steps made by a student.

Zinn [44] writes strategies as Prolog programs, in which rules and strategies ('task models') are intertwined. His system gives detailed feedback and supports buggy rules, with results similar to those we obtain. We believe that explicitly modeling the strategy language makes it easier to specify strategies, which in Zinn's approach have to be

programmed directly in PROLOG. Furthermore, the implementation of our strategy language provides us with lots of possibilities for giving feedback that can be reused for all other domains. Some of these kinds of feedback, such as strategy unfolding, can only be given if the strategy is explicitly specified.

## 8 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. The formulation of a strategy as a CFG allows us to automatically calculate several kinds of feedback to students incrementally solving exercises. Languages for modeling 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 Sect. 5 we have shown how strategies can be used to report advanced feedback.

We have also presented the 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 shown how our implementation of a recognizer can be used to automatically calculate several kinds of feedback.

### 8.1 Current Status and Future Work

At the present time we have defined (and experimented with) various strategies in different domains. Examples from domains in which we have invested most effort have been included in the paper: bringing a proposition in DNF, and reducing a matrix using Gaussian elimination. In addition to Gaussian elimination, more strategies have been developed for linear algebra, such as solving a linear system of equations (with or without converting to a matrix), and orthogonalizing a set of vectors (Gram-Schmidt). These strategies have been tested with students on a small scale. Lodder et al. [33] report on tests with our own tool that supports bringing a proposition in DNF. Corbalan and Paas [15] show that providing feedback on the strategic level in interactive linear algebra exercises improves far transfer [34]: students that received feedback on the strategic level did better in advanced exercises. The linear algebra exercises are provided via MathDox [14,21], to which we have created a binding.

A spin-off from the logical domain is a strategy for relation algebra, in which the goal is to reach a conjunctive normal form. Relation algebra is used for a course on business rules at our university. In our experience, students in this course struggle with the formal nature of relation algebra, and have difficulties with manipulating terms correctly. It is our hope that an exercise assistant on relation algebra supports the student in understanding the course material. A proto-type implementation is available, although it currently ignores multiplicities of relations.

A recent focus is on strategies for bridging courses in mathematics. We have formulated strategies for linear equations, quadratic equations, and equations of a higher degree. Interestingly, there are several well-known strategies around for solving an exercise class. For instance, the equation $3(x + 1) = 6$ can be solved by first applying a rule for distribution, or by observing that $(x + 1)$ must be equal to 2. Our strategy language is expressive enough to train students to select the "right" approach for a given exercise. We expect to develop many more strategies at this level within the Dutch Surf NKBW project, and the European eContentPlus Math-Bridge project. The goal of both of these projects is to provide mathematical content to bridge the gap between high-schools and universities.

To make our strategies available, we have created bindings with two more external tools: the DWO [18], and Active-Math [21]. Since the number of external tools to which we create bindings is increasing, we need to standardize the protocol for providing feedback.

We have started to explore the construction of strategies for programming [19,3], and the first results are encouraging. The only additional concept we need, compared to the mathematical domains, is refinement rules. A refinement rule refines a program by taking a small step towards completing the program. Using our strategy language for specifying programming exercises offers the possibility to efficiently calculate feedback while incrementally developing a program, and to significantly reduce the effort in adding new programming exercises to an intelligent tutor for programming. Strategies for solving programming exercises is ongoing research.

We want to apply our ideas to several other domains. As mentioned earlier, we think we can use our approach to support learning theorem proving and computer programming. We are also interested in domains with less structure such as software modeling, and maybe even serious games in which students have to apply a backtracking strategy, or 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. Anderson, J.R.: Rules of the Mind. Lawrence Erlbaum Associates, Hillsdale (1993)
2. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: lessons learned. J. Learn. Sci. **4**(2), 167–207 (1995)
3. Anderson, J.R., Skwarecki, E.: The automated tutoring of introductory computer programming. Commun. ACM **29**(9), 842–849 (1986)
4. Beeson, M.J.: Design principles of Mathpert: software to support education in algebra and calculus. In: Kajler, N. (ed.) Computer-Human Interaction in Symbolic Computation, pp. 89–115. Springer, New York (1998)
5. Boon, P., Drijvers, P.: Algebra en applets, leren en onderwijzen (algebra and applets, learning and teaching, in Dutch). http://www.fi.uu.nl/publicaties/literatuur/6571.pdf (2005)
6. Bouwers, E.: Improving automated feedback—a generic rule-feedback generator. Master's thesis, Department of Information and Computing Sciences, Utrecht University (2007)
7. Brown, J.S., Burton, R.R.: Diagnostic models for procedural bugs in basic mathematical skills. Cogn. Sci. **2**, 155–192 (1978)
8. Brown, J.S., VanLehn, K.: Repair theory: a generative theory of bugs in procedural skills. Cogn. Sci. **4**, 379–426 (1980)
9. Bundy, A.: The Computer Modelling of Mathematical Reasoning. Academic Press, San Diego (1983)
10. Bundy, A.: The use of explicit plans to guide inductive proofs. In: Conference on Automated Deduction, pp. 111–120 (1988)
11. Bundy, A.: A critique of proof planning. In: Computat. Logic (Kowalski Festschrift), LNAI, vol. 2408 (2002)
12. Chaachoua, H., et al.: Aplusix, a learning environment for algebra, actual use and benefits. In: ICME 10: 10th International Congress on Mathematical Education, 2004. Retrieved http://www.itd.cnr.it/telma/papers.php, May 2008
13. Cohen, A., Cuypers, H., Jibetean, D., Spanbroek, M.: Interactive learning and mathematical calculus. In: Mathematical Knowledge Management (2005)
14. Cohen, A., Cuypers, H., Barreiro, E.R., Sterk, H.: Interactive mathematical documents on the web. In: Algebra, Geometry and Software Systems, pp. 289–306. Springer, New York (2003)
15. Corbalan, G., Paas, F.: Learning and motivational effects of different forms of feedback in linear algebra problems. Paper presented at the math tutoring: tools and feedback symposium organized by the surf foundation and the joining educational mathematics network (JEM), Heerlen, The Netherlands (2008)
16. Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. Electron. Notes Theor. Comput. Sci. **174**(1), 17–34 (2007)
17. Erev, I., Luria, A., Erev, A.: On the effect of immediate feedback. http://telem-pub.openu.ac.il/users/chais/2006/05/pdf/e-chais-erev.pdf (2006)
18. Freudenthal Institute: Digital math environment. http://www.fi.uu.nl/dwo (2004)
19. Gerdes, A., Heeren, B., Jeuring, J.: Constructing strategies for programming. In: Cordeiro, J., Shishkov, B., Verbraeck, A., Helfert, M. (eds.) Proceedings CSEDU'09: 1st International Conference on Computer Supported Education, pp. 65–72 (2009)
20. Gerdes, A., Heeren, B., Jeuring, J., Stuurman, S.: Feedback Services for Exercise Assistants. In: Remenyi, D. (ed.) The Proceedings of the 7th European Conference on e-Learning, pp. 402–410. Academic Publishing Limited (2008)

21. Goguadze, G., Palomo, A.G., Melis, E.: Interactivity of exercises in ActiveMath. In: International Conference on Computers in Education, ICCE 2005 (2005)
22. Heeren, B., Jeuring, J.: Recognizing strategies. In: Middeldorp, A. (ed.) WRS 2008: Reduction Strategies in Rewriting and Programming, 8th International Workshop (2008)
23. Hennecke, M.: Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen (in German). PhD thesis, Hildesheim University, Fortschritt-Berichte VDI Reihe 10, Informatik/Kommunikationstechnik; 605. VDI, Düsseldorf (1999)
24. Hinze, R., Jeuring, J., Löh, A.: Comparing approaches to generic programming in Haskell. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) Datatype-Generic Programming, LNCS, vol. 4719, pp. 72–149. Springer, New York (2007)
25. Horacek, H., Wolska, M.: Handling errors in mathematical formulas. In: Ikeda, M., Ashley, K., Chan, T.-W. (eds.) ITS 2006, LNCS, vol. 4053, pp. 339–348. Springer, New York (2006)
26. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. **28A**(4), 17 (1996)
27. Hutton, G.: Higher-order functions for parsing. J. Funct. Program. **2**(3), 323–343 (1992)
28. Issakova, G.: Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. PhD thesis, University of Tartu (2007)
29. Jones, S.P., 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/
30. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical approach to generic programming. ACM SIGPLAN Not. **38**(3), 26–37. TLDI'03 (2003)
31. Lämmel, R., Visser, E., Visser, J.: The Essence of Strategic Programming, 18 pp.; Draft; Available http://www.cwi.nl/~ralf. Accessed 15 Oct 2002
32. Lodder, J., Jeuring, J., Passier, H.: An interactive tool for manipulating logical formulae. In: Manzano, M., Lancho, B.P., Gil, A. (eds.) Proceedings of the Second International Congress on Tools for Teaching Logic (2006)
33. Lodder, J., Passier, H., Stuurman, S.: Using ideas in teaching logic, lessons learned. In: Computer Science and Software Engineering, International Conference, vol. 5, pp. 553–556 (2008)
34. National Research Council: How People Learn—Brain, Mind, Experience, and School. National Academy Press (2000)
35. Passier, H., Jeuring, J.: Feedback in an interactive equation solver. In: Seppälä, M., Xambo, S., Caprotti, O. (eds.) Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006, pp. 53–68. Oy WebALT Inc. (2006)
36. Paulson, L.C.: ML for the Working Programmer, 2nd edn. Cambridge University Press, Cambridge (1996)
37. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: ICFP '07, pp. 141–152, New York (2007)
38. Sweller, J., van Merriënboer, J.J.G., Paas, F.: Cognitive architecture and instructional design. Educ. Psychol. Rev. **10**, 251–295 (1998)
39. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Meijer, E., Sheard, T. (eds.) Advanced Functional Programming, LNCS, vol. 1129, pp. 184–207. Springer, New York (1996)
40. van Ditmarsch, H.: Logic software and logic education. http://www.cs.otago.ac.nz/staffpriv/hans/logiccourseware.html (2009)
41. van Merriënboer, J.J.G., Jelsma, O., Paas, F.G.W.C.: Training for reflective expertise: a four-component instructional design model for complex cognitive skills. Educ. Technol. Res. Dev. **40**(2), 23–43 (1992)
42. VanLehn, K.: Mind Bugs—The Origins of Procedural Misconceptions. MIT Press, Cambridge (1990)
43. Visser, E., Benaissa, Z.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: ICFP '98, pp. 13–26 (1998)
44. Zinn, C.: Supporting tutorial feedback to student help requests and errors in symbolic differentiation. In: Ikeda, M., Ashley, K., Chan, T.-W. (eds.) ITS 2006, LNCS, vol. 4053, pp. 349–359. Springer, New York (2006)