Specifying Strategies for Exercises

Bastiaan Heeren

Johan Jeuring

Arthur van Leeuwen

Alex Gerdes

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2008-001

www.cs.uu.nl

ISSN: 0924-3275

Specifying Strategies for Exercises

Bastiaan Heeren, Johan Jeuring, Arthur van Leeuwen, and Alex Gerdes

School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
{bhr,jje,ale}@ou.nl

Abstract. The feedback given by e-learning tools that support incrementally solving problems in mathematics, logic, physics, etc. is limited, or laborious to specify. In this paper we introduce a language for specifying strategies for solving exercises. This language makes it easier to automatically calculate feedback when users make erroneous steps in a calculation. Although we need the power of a full programming language to specify strategies, we carefully distinguish between context-free and non-context-free sublanguages of our strategy language. This separation is the key to automatically calculating all kinds of desirable feedback.

1 Introduction

Tools like Aplusix [9], LeActiveMath [13], and MathPert [4]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 latter kind of feedback.

This paper. 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.

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 discusses the 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

Strategies. 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 latter kind of knowledge 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 describing how a simple arithmetic expression with constants, $+$ and $-$ can be simplified, to very complex, describing a complicated procedure for solving an exercise from linear algebra.

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. Using a computer for practicing strategies has several advantages. An e-learning system or an intelligent tutoring system can:

- immediately tell a student whether or not a solution to an exercise is correct,
- explain the most likely error when an answer is incorrect,
- maintain a model of the student, and use this model to give specialized feedback to a student,
- generate an infinite amount of exercises,
- give hints when a student is stuck, etc.

There exist many variants of tools that support solving exercises. 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 so 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. [19] 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 feedback that is given 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 [14] 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), about 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 rule from itself), and about the strategy for Gaussian elimination of a matrix (which we will give in the appendix).

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 [17] 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 analyse 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.
- Based on the position on the path from the starting point to the solution of an exercise we create a progress bar.

- If a student enters a wrong final answer, we 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 [24]. Using the basic constructs from the strategy language, we can create something that looks like 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 [27], but also for e-learning tools [16]. Some work has been done on trying to explain errors made by students on the level of rewrite rules [14, 20, 23, 5].

Already around 1980, but also later, VanLehn et al. [6, 7, 28], 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 [28], 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. Neither VanLehn nor Anderson use parsing for the language for procedures to automatically calculate feedback.

Zinn [30] 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. In the appendix we give a more involved strategy for Gaussian elimination on matrices.

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:

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

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.

Basic Rules:

Constants: AndTrue : $p \wedge T = p$ OrTrue : $p \vee T = T$ NotTrue : $\neg T = F$
AndFalse : $p \wedge F = F$ OrFalse : $p \vee F = p$ 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: DeMorganAnd : $\neg(p \wedge q) = \neg p \vee \neg q$ NotNot : $\neg\neg p = p$
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: OrTaut : $p \vee \neg p = T$ EquivTaut : $p \leftrightarrow p = T$
ImplTaut : $p \rightarrow p = T$

Contradictions: AndContr : $p \wedge \neg p = F$ EquivContr : $p \leftrightarrow \neg p = F$

Fig. 1: Rules for logical expressions

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 was applied. Note that these rules should not appear in strategies, since that would invalidate the strategy.

The idea of formulating buggy rules can easily be extended to buggy strategies. Such a strategy helps to recognize common procedural mistakes, in which case we can 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 $r(a)$ to denote 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. In fact, we can view a strategy as a grammar where the rules form the alphabet of the language.

- **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.

– **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 can apply a strategy s to a term a, written $s(a)$, just as we can apply some rule. We make the informal description of the presented combinators precise by giving a formal definition for each of the combinators:

$$
\begin{array}{ll}
(s \mathbin{<\!\star\!>} t)(a) & = \{\, t(b) \mid b \leftarrow s(a) \,\} \\
(s \mathbin{<|>} t)(a) & = s(a) \cup t(a) \\
\mathit{succeed}(a) & = \{a\} \\
\mathit{fail}(a) & = \emptyset \\
(\mathit{label}\ \ell\ s)(a) & = s(a)
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{many}\ \ s = (s \mathbin{<\!\star\!>} \mathit{many}\ s) \mathbin{<|>} \mathit{succeed} \\
\mathit{many1}\ s = s \mathbin{<\!\star\!>} \mathit{many}\ s \\
\mathit{option}\ \ s = s \mathbin{<|>} \mathit{succeed}
\end{array}
$$

Observe the recursion in the definition of *many*. Depending on the implementation one prefers, the *many* combinator results in an infinite strategy (which is not at all a problem in a lazy programming language such as Haskell), or this combinator gets a special treatment and is implemented as a primitive. 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:

$$
\begin{array}{lll}
(\mathit{not}\ s)(a) = \{a\} & & \mathit{if}\ s(a) = \emptyset \\
\qquad\qquad = \emptyset & & \mathit{otherwise}
\end{array}
$$

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.

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

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:

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

Using generic programming techniques [15], 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).

$$
\mathit{somewhere}\ s = s <|> \mathit{once}\ (\mathit{somewhere}\ s)
$$

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

$$
\begin{array}{ll}
\mathit{bottomUp}\ s = \mathit{once}\ (\mathit{bottomUp}\ s) \triangleright s \\
\mathit{topDown}\ \ s = s \triangleright \mathit{once}\ (\mathit{topDown}\ s)
\end{array}
$$

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:

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

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

$$
\begin{aligned}
\mathit{dnfStrategy1} &= \mathit{repeat}\ (\mathit{somewhere}\ \mathit{basicRules}) \\
\mathit{dnfStrategy2} &= \mathit{label}\ \texttt{"step 1"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants})) \\
&\quad \mathbin{<\star>} \mathit{label}\ \texttt{"step 2"}\ (\mathit{repeat}\ (\mathit{bottomUp}\ \mathit{definitions})) \\
&\quad \mathbin{<\star>} \mathit{label}\ \texttt{"step 3"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{negations})) \\
&\quad \mathbin{<\star>} \mathit{label}\ \texttt{"step 4"}\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{distribution}))
\end{aligned}
$$

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 postpone the discussion and give a more elegant definition after the reflections.

4.6 Reflections

Is the set of strategy combinators complete? Not really, although we hope to have convinced the reader how easily the language can be 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 strongly believe that our combinators are sufficient for specifying the kind of strategies that are needed in interactive exercise assistants that aim at providing intelligent feedback. Our language is very similar to strategic programming languages such as Stratego [29, 22], and very similar languages are used in parser combinator libraries [18, 27], boiler-plate libraries [21], workflow applications [25], and data-conversion libraries [12], which suggests that our library could serve as a firm basis for strategy specifications.

One strategy combinator that we have not yet tackled is $s \mathbin{<||>} t$, which applies the strategies s and t in parallel, i.e., interleaving steps from s with steps from t. Although this combinator would not allow us to define more strategies,

it does help to specify certain strategies more concisely. We realize that applying strategies in parallel might not be ideal when trying to teach procedural skills.

With parallel strategy combinators, we can give a concise definition for the third DNF strategy, in which we reuse our second strategy. We assume to have the left-biased variant of the parallel combinator at our disposal, for which we will write $s \ \|\!\rhd\ t$. Similar to the left-biased choice operator ($\triangleright$), this strategy applies a rule from s if possible. We first define a new and reusable combinator, followed by a definition for the strategy:

$$\begin{aligned} &\mathit{whenever}\ s\ t\ = \mathit{repeat}\ s\ \|\!\rhd\ t \\ &\mathit{dnfStrategy3} = \mathit{whenever}\ ((\mathit{tautologies} <\!|\!> \mathit{contradictions}) <\!\star\!>\ \mathit{step1}) \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{dnfStrategy2} \end{aligned}$$

In the above definition, *step1* is equal to $\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants})$.

In Appendix A, we present a complete strategy that implements the Gaussian elimination algorithm. This example is more involved, and shows, amongst others, how rules can be parameterized, and how to maintain additional information in a context while running a strategy.

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.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.

Feedback after a step. After each step a student performs, we check whether or not this step is valid according to the strategy. 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. Hence, this is essentially a parsing problem. As soon as we detect that a student no longer follows the strategy, we have several opportunities to react on this. 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 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.

Progress. Given an exercise and a strategy for solving the exercise, we 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 a 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 *true* and *false* in the logical expression. After completing this part of the exercise, we ask to solve the remaining part of the exercise.

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 the part of the strategy that is not context-free, we specify generators for generating the necessary variables and arguments. For example, when doing Gaussian elimination, our strategy specifies which rows have to be added or swapped when asked for a hint. An alternative possibility for hints is to also 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. 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 [26]. 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 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 specified strategy, but also with respect to known buggy strategies.

6 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.

We have several plans for the future. We hope to create bindings of our feedback service with more existing tools, such as LeActiveMath. For this purpose, we need to standardize the protocol for providing feedback. Also, we want to apply our ideas to domains with less structure, such as computer programming [3], software modelling, and maybe even serious games in which students have to cooperate to achieve a certain goal. Our tools are going to be used in several courses during 2008. We will collect data from the experiments, and analyze and report on the results.

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`, January 2007.
10. Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.
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. G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. In *International Conference on Computers in Education, ICCE 2005*, 2005.

14. 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.
15. 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, International Spring School, SSDGP 2006, Nottingham, UK April 24-27, 2006, Revised Lectures*, volume 4719 of *LNCS*, pages 72–149. Springer-Verlag, 2007.
16. 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.
17. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.
18. Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.
19. Anan Erev Ido Erev, Adi Luria. On the effect of immediate feedback, May 2006. Retrieved from `http://telem-pub.openu.ac.il/users/chais/2006/05/pdf/e-chais-erev.pdf`, December 2006.
20. Marina Issakova. *Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment*. PhD thesis, University of Tartu, 2007.
21. 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.
22. Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. 18 p.; Draft; Available at `http://www.cwi.nl/~ralf`, October15 2002.
23. 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.
24. 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/`.
25. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 141–152, New York, NY, USA, 2007. ACM.
26. J. Sweller, J.J.G. van Merriënboer, and F. Paas. Cognitive architecture and instructional design. *Educational Psychology Review*, 10:251–295, 1998.
27. S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.
28. Kurt VanLehn. *Mind Bugs – The Origins of Procedural Misconceptions*. MIT Press, 1990.
29. Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
30. 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.

A Gaussian elimination

In this appendix we present the implementation of a strategy to bring a matrix into reduced echelon form: this procedure is also known as Gaussian elimination. Functions to manipulate matrices have been omitted, but most of the interesting parts are given. We want to emphasize that the given code is a valid and executable Haskell program.

A.1 Preliminaries

A matrix is represented by the abstract data type *Matrix a*, where the type variable *a* is the type of the entries (typically rational numbers). Besides a matrix, we maintain one variable for the *j*-th column and the number of rows that is covered. This is the context in which we perform Gaussian elimination:

$$\textbf{data}\ \mathit{MatrixInContext}\ a = \mathit{MIC}\{\,\mathit{matrix} :: \mathit{Matrix}\ a, \mathit{columnJ} :: \mathit{Int}, \mathit{covered} :: \mathit{Int}\,\}$$

We define one helper-function that returns the matrix without its covered rows:

$$\begin{aligned}&\mathit{subMatrix} :: \mathit{MatrixInContext}\ a \to \mathit{Matrix}\ a\\ &\mathit{subMatrix}\ c = \mathit{makeMatrix}\ \ \$\ \ \mathit{drop}\ (\mathit{covered}\ c)\ \ \$\ \ \mathit{rows}\ \ \$\ \ \mathit{matrix}\ c\end{aligned}$$

Gaussian elimination revolves around three elementary operations on rows:

$$\begin{aligned}&\mathit{rowScale} &&:: \mathit{Num}\ a \Rightarrow \mathit{Int} \to &&a \to \mathit{Transformation}\ (\mathit{MatrixInContext}\ a)\\ &\mathit{rowExchange} &&:: \quad\ \mathit{Int} \to \mathit{Int} \to &&\mathit{Transformation}\ (\mathit{MatrixInContext}\ a)\\ &\mathit{rowAdd} &&:: \mathit{Num}\ a \Rightarrow \mathit{Int} \to \mathit{Int} \to a \to &&\mathit{Transformation}\ (\mathit{MatrixInContext}\ a)\end{aligned}$$

These three functions all return a transformation. Although we leave this type undefined, it is useful to have the following type synonym in mind:

$$\textbf{type}\ \mathit{Transformation}\ a = a \to \mathit{Maybe}\ a$$

The three transformation functions work as follows: *rowScale i a* multiplies row *i* by a non-zero number *a*, *rowExchange i j* interchanges the two rows *i* and *j*, and *rowAdd i j a* adds *a* times row *j* to row *i*. Both *rowScale* and *rowAdd* require that the entries of the matrix are values supporting the numerical operations $(+)$ and $(*)$, hence the type constraint *Num a* in their type signatures. Having defined a context and three elementary operations, we are now ready to define a strategy, and we will do so in a top-down fashion.

A.2 Strategy

The strategy consists of a forward pass, followed by a backward pass:

$$\begin{aligned}&\mathit{toReducedEchelon} :: \mathit{Fractional}\ a \Rightarrow \mathit{Strategy}\ (\mathit{MatrixInContext}\ a)\\ &\mathit{toReducedEchelon} = \mathit{label}\ \texttt{"Gaussian elimination"}\ \$\\ &\quad \mathit{forwardPass} \mathbin{<\!\star\!>} \mathit{backwardPass}\end{aligned}$$

The forward pass is again a strategy, consisting of five steps. These steps have to be repeated until this is no longer possible:

```
forwardPass :: Fractional a ⇒ Strategy (MatrixInContext a)
forwardPass = label "Forward pass" $
  repeat  $  label "Find j-th column"      ruleFindColumnJ
         <*> label "Exchange rows"         (try ruleExchangeNonZero)
         <*> label "Scale row"             (try ruleScaleToOne)
         <*> label "Zeros in j-th column" (repeat ruleZerosFP)
         <*> label "Cover up top row"      ruleCoverRow
```

Each of the five steps uses one *ad-hoc* rule: *ruleFindColumnJ*, *ruleExchangeNonZero*, etc. These rules will be discussed in detail in the next section. Observe that the strategy combinators *try* and *repeat* are used in some of these steps. The type class constraint *Fractional a* arises from *ruleScaleToOne* which requires division $(/)$ on the elements of the matrix.

The structure of the backward pass is quite similar to the forward pass:

```
backwardPass :: Fractional a ⇒ Strategy (MatrixInContext a)
backwardPass = label "Backward pass" $
  repeat  $  label "Uncover row" ruleUncoverRow
         <*> label "Sweep"       (repeat ruleZerosBP)
```

A.3 Rules

We now define the seven rules that are mentioned in the forward and the backward pass. We will first consider the rules that change the context (i.e., column j and the number of covered rows) without changing the matrix. The smart constructor *makeSimpleRule* turns a function of type $a \to Maybe\ a$ into a *Rule* a[1]. Each rule has a unique name.

```
ruleFindColumnJ :: Num a ⇒ Rule (MatrixInContext a)
ruleFindColumnJ = minorRule $ makeSimpleRule "FindColumnJ" $
  λc → do let cols = columns (subMatrix c)
          j    ← findIndex (any (≠ 0)) cols
          return c{columnJ = j}
```

Column j becomes the first column with a non-zero entry (ignoring the covered rows). We tag this rule as *minor* because it only changes the context. The two other minor rules are for covering and uncovering one row:

```
ruleCoverRow   :: Rule (MatrixInContext a)
ruleUncoverRow :: Rule (MatrixInContext a)
```

[1] In our actual implementation we have overloaded all strategy combinators, for instance, <*> and *repeat*. Hence, we don't have to lift rules into strategies. Both *forwardPass* and *backwardPass* are properly typed.

These functions simply increment and decrement the counter *covered* that is stored in the context by one.

All the remaining rules operate on the matrix and use the elementary row operations for this. These operations are, however, parameterized, and thus we still have to supply the required arguments. For this purpose, we introduce the combinator *apply*:

$$apply :: (a \to Transformation\ b) \to (b \to Maybe\ a) \to Transformation\ b$$

For each parameterized transformation we need a function that given a context (of type b) returns the argument (of type a), or that fails. For convenience, we define two derived combinators for transformations with multiple arguments:

$$\begin{aligned} &apply2\ f = apply\ (\lambda(x,y) \to f\ x\ y)\\ &apply3\ f = apply\ (\lambda(x,y,z) \to f\ x\ y\ z)\end{aligned}$$

We now consider the steps of the forward pass. The following rule exchanges (if necessary) the first row with another row that has a non-zero entry in the j-th column:

```
ruleExchangeNonZero :: Num a ⇒ Rule (MatrixInContext a)
ruleExchangeNonZero =
  makeRule "ExchangeNonZero" $ apply2 rowExchange $
    λc → do let col = column (columnJ c) (subMatrix c)
            i   ← findIndex (≢ 0) col
            return (covered c, i + covered c)
```

The function *makeRule* turns a transformation into a rule, just like *makeSimpleRule*. Next, we scale the first row to make the leading entry 1. This rule fails if this is already the case.

```
ruleScaleToOne :: Fractional a ⇒ Rule (MatrixInContext a)
ruleScaleToOne = makeRule "ScaleToOne" $ apply2 rowScale $
  λc → do let pv = entry (0, columnJ c) (subMatrix c)
          return (covered c, 1 / pv)
```

The next rule introduces zeros in the j-th column during the forward pass. We extract column j without the first row, and search for a row with a non-zero entry. We then compute and return the right value for v such that after addition we get a zero in this row.

```
ruleZerosFP :: Num a ⇒ Rule (MatrixInContext a)
ruleZerosFP =
  makeRule "Introduce zeros (forward pass)" $ apply3 rowAdd $
    λc → do let col = drop 1 $ column (columnJ c) (subMatrix c)
            i   ← findIndex (≢ 0) col
            let v   = negate (col !! i)
            return (i + covered c + 1, covered c, v)
```

The last rule introduces zeros during the backward pass, and is used multiple times after an application of *ruleUncoverRow*. First we determine the pivot position in the first uncovered row: let this be j. Then we search for a row that has a non-zero entry in the j-th column (variable k), and we calculate the right value for v to obtain a zero. Note that k can be one of the covered rows.

```
ruleZerosBP :: Num a ⇒ Rule (MatrixInContext a)
ruleZerosBP =
  makeRule "Introduce zeros (backward pass)" $ apply3 rowAdd $
    λc → do let j   = length $ takeWhile (≡ 0) $ row 0 $ subMatrix c
                col = column j (matrix c)
            k       ← findIndex (≢ 0) col
            let v   = negate (col !! k)
            return (k, covered c, v)
```

B Simplifying arithmetic expressions

In this appendix we present an implementation of another strategy: simplify an arithmetic expression, including fractions. Instead of giving the entire implementation, we only highlight interesting parts. The complete implementation can be found at the location mentioned in appendix A. We want to stress that the given strategy is a *possible* strategy to bring an expression in a simplified form. There may be other and perhaps more efficient strategies. Furthermore, we limit ourselves to *linear* expressions. Before defining the strategy, we briefly introduce the domain of arithmetic expressions.

B.1 Domain description

An arithmetic expression consists of additions $+$, subtractions $-$, multiplications $*$, divisions $/$, constants, and variables. Sub-expressions are surrounded by parentheses. Our arithmetic expressions use constants of the same type, in our case integers. An example expression:

$$((7+1)*(3-3))\ /\ (-2*1)+1-0+x$$

The arithmetic expression domain can be described by the following grammar:

$$\begin{aligned} \mathit{AExpr} &::= \mathit{Var} \mid \mathit{Con} \mid \mathit{AExpr} * \mathit{AExpr} \mid \mathit{AExpr}\ /\ \mathit{AExpr} \\ &\quad \mid \mathit{AExpr} + \mathit{AExpr} \mid \mathit{AExpr} - \mathit{AExpr} \mid (\mathit{AExpr}) \\ \mathit{Var} &::= \{a \ldots z\}^{+} \\ \mathit{Con} &::= \{-\}^{?}\{0 \ldots 9\}^{+} \end{aligned}$$

The given grammar can be modeled in Haskell with the following datatype:

```
data AExpr = Var String
           | Con Integer
```

```
  | AExpr :*: AExpr
  | AExpr :/: AExpr
  | AExpr :+: AExpr
  | AExpr :-: AExpr
```

Note that this representation allows to construct invalid expressions[2]. Instead of devising a more complex datatype that prohibits such expressions, we use a *generator*[3] that only produces valid expressions. The representation in *AExpr* of the example given above is:

```
((Con 7 :+: Con 1) :*: (Con 3 :-: Con 3)) :/:
  (Con (-2) :*: Con 1) :+: Con 1 :-: Con 0 :+: Var "x"
```

In order to determine if a given step is valid, we need to check if the given term and the previous term are *semantically* the same. We need to be able to compare arithmetic expressions for equivalence. The implementation of this equivalence function is not trivial. The algorithm we use rewrites expressions to a 'normal' form and compares them *syntactically*. To rewrite an expression in a 'normal' form, the coefficient of every variable is calculated. Next the variables, along with their coefficients, are ordered alphabetically, followed by the result of the calculation of all constants (which may be a fraction).

B.2 Strategy

The strategy to simplify an arithmetic expression consists of three top-level steps, which are repeated until they cannot be applied anymore. The top-level steps are: eliminate all zeros, including multiplications by zero, followed by eliminating all units, including additions with zero, and a calculation step, e.g. add two constants. The definition of this strategy using the strategy combinators is as follows:

```
toSimpleExpr :: NamedStrategy AExprInContext
toSimpleExpr  =  label "Simplify expression" $ repeatNS $
                 label "Eliminate zeros" eliminateZeros
              <*> label "Eliminate units" eliminateUnits
              <*> label "Calculatie"      calculate
```

Let us examine the most interesting step, *calculate*, in more detail:

```
calculate :: Strategy AExprInContext
calculate  =  somewhere $
              ruleMul <|> ruleDiv <|> ruleAdd <|> ruleSub
          <|> ruleGCD
          <|> ruleDistMul
          <|> calcFrac
```

[2] Like non-linear expressions and expressions containing a division by zero

[3] The generator makes use of QuickCheck [10]

The calculate step consists of rules for the trivial operators (like addition and multiplication), a rule for dividing both components of a fraction by their greatest common divisor, a rule for distributing multiplication over addition and subtraction and a sub-strategy called *calcFrac*, all combined with the *choice* combinator.

Notice that this step is not repeated, just one rule is applied somewhere, using the *somewhere* combinator. After applying a calculation rule, the expression should be freed from units and zeros again before performing the next calculation step.

The sub-strategy, *calcFrac*:

$$
\begin{aligned}
&\mathit{calcFrac} :: \mathit{Strategy}\ \mathit{FracInContext}\\
&\mathit{calcFrac} \;=\; \mathit{ruleCommonDenom}\\
&\qquad <\star> (\mathit{ruleAddFrac} <|> \mathit{ruleSubFrac})\\
&\qquad <\star> \mathit{ruleGCD}
\end{aligned}
$$

consists of the following sequential steps: rewrite fractions to have a common denominator, add or subtract two fractions, and reduce the resulting fraction in the final step. The rules in this sub-strategy will be examined in more detail in the next section.

The next derivation shows a simplification that follows the given strategy:

$$
\begin{aligned}
&\quad ((7+1)*(3-3))\ /\ (-2*1)+1-0+x\\
&\Rightarrow ((7+1)*(3-3))\ /\ (-2*1)+1+x\\
&\Rightarrow ((7+1)*(3-3))\ /\ -2+1+x\\
&\Rightarrow ((7+1)*(3-3))\ /\ -2+1+x\\
&\Rightarrow (8*(3-3))\ /\ -2+1+x\\
&\Rightarrow (8*0)\ /\ -2+1+x\\
&\Rightarrow 0\ /\ -2+1+x\\
&\Rightarrow 0+1+x\\
&\Rightarrow 1+x
\end{aligned}
$$

B.3 Rules

In this section we will highlight some rules that are used in the strategy for simplifying arithmetic expressions. We will start with a simple rule, which is used in the *eliminateUnits* step, that removes an addition with zero. Here we use the *makeRuleList* function, which turns a list of transformations into a rule:

$$
\begin{aligned}
&\mathit{ruleUnitAdd} :: \mathit{AExprRule}\\
&\mathit{ruleUnitAdd} = \mathit{makeRuleList}\ \texttt{"UnitAdd"}\\
&\quad [(x :+: \mathit{Con}\ 0) \;=\; x\\
&\quad ,(\mathit{Con}\ 0 :+: x) \;=\; x\\
&\quad]
\end{aligned}
$$

A more elaborate rule is *ruleCommonDenom*, which transforms the denominators of two fractions or a fraction and a constant, to a common one. This function uses pattern matching to select the correct transformation.

```
ruleCommonDenom :: AExprRule
ruleCommonDenom = makeSimpleRule "CommonDenom" commonDenom
where
  commonDenom (Con x :/: Con y :+: Con v :/: Con w)
    | y ≢ w     = return $ Con (x * w) :/: Con (y * w) :+:
                           Con (v * y) :/: Con (y * w)
    | otherwise = Nothing
  commonDenom (Con x :/: Con y :−: Con v :/: Con w)
    | y ≢ w     = return $ Con (x * w) :/: Con (y * w) :−:
                           Con (v * y) :/: Con (y * w)
    | otherwise = Nothing
  commonDenom (Con x :+: Con v :/: Con w)
    = return $ Con (x * w) :/: Con w :+: Con v :/: Con w
  commonDenom (Con x :/: Con y :+: Con v)
    = return $ Con x :/: Con y :+: Con (v * y) :/: Con y
  commonDenom (Con x :−: Con v :/: Con w)
    = return $ Con (x * w) :/: Con w :−: Con v :/: Con w
  commonDenom (Con x :/: Con y :−: Con v)
    = return $ Con x :/: Con y :−: Con (v * y) :/: Con y
  commonDenom _ = Nothing
```

In the sub-strategy *calcFrac* the next rule, a choice between *ruleAddFrac* and *ruleSubFrac*, is applied after the *ruleCommonDenom* rule has been applied. It adds (subtracts) two fractions, which results in a single fraction.

```
ruleAddAExpr :: AExprRule
ruleAddAExpr = makeSimpleRule "AddAExpr" addAExpr
where
  addAExpr (Con x :/: Con y :+: Con v :/: Con w)
    | y ≡ w     = return $ Con (x + v) :/: Con w
    | otherwise = Nothing
  addAExpr _    = Nothing
```

The last rule of the *calcFrac* strategy transforms a fraction into its reduced form. If the fraction is not already reduced, the *greatest common divisor* is calculated, which is used to divide both the numerator and denominator.

```
ruleGCD :: AExprRule
ruleGCD = makeSimpleRule "GCD" rGCD
where
  rGCD (Con x :/: Con y)
      | a ≡ 1     = Nothing
      | otherwise = return $ Con (x `div` a) :/: Con (y `div` a)
    where
      a = gcd x y
  rGCD _ = Nothing
```