Properties of Exercise Strategies

Alex Gerdes Bastiaan Heeren
School of Computer Science
Open Universiteit Nederland
Heerlen, The Netherlands
age@ou.nl bhr@ou.nl

Johan Jeuring
Department of Information and Computing Sciences
Utrecht Univeristy
Utrecht, The Netherlands
johanj@cs.uu.nl

Mathematical learning environments give domain-specific and immediate feedback to students solving a mathematical exercise. Based on a language for specifying strategies, we have developed a feedback framework that automatically calculates semantically rich feedback. We offer this feedback functionality to mathematical learning environments via a set of web services. Feedback is only effective when it is precise and to the point. The tests we have performed give some confidence about the correctness of our feedback services. To increase confidence in our services, we explicitly specify the properties our feedback services should satisfy, and, if possible, prove them correct. For this, we give a formal description of the concepts used in our feedback framework services. The formalisation allows us to reason about these concepts, and to state a number of desired properties of the concepts. Our feedback services use exercise descriptions for their instances on domains such as logic, algebra, and linear algebra. We formulate requirements these domain descriptions should satisfy for the feedback services to react as expected.

1 Introduction

We use strategies to calculate *semantically rich feedback* for students solving a mathematical exercise [9]. For example, we can calculate a hint, or show a complete derivation for a number of mathematical domains, such as propositional logic, linear algebra, and arithmetic. A strategy captures expert knowledge about how to solve a particular problem. It describes which steps a student can take to solve an exercise, and in what order. When a student solves an exercise stepwise, we can check whether or not a step follows the strategy.

We have developed an embedded domain-specific strategy language in which we can specify strategies, and designed a feedback framework on top of it. This framework is used to give detailed feedback to interactive mathematical environments such as ActiveMath [14], the Freudenthal Digital Mathematics Environment [3], and our own tool for rewriting logic expressions [13]. A set of feedback services [7] gives mathematical environments access to our feedback functionality.

Feedback tops the list of factors leading to good learning [2], but it is only effective when it is precise and to the point. Feedback messages should not mislead a student practicing an exercise. Therefore, we want to ensure that the given feedback is to the point and relevant. The software we have developed is augmented with numerous (unit) tests to test correct behaviour. However, a formal definition of the concepts used, such as the strategy language, exercises, and the services, contributes to the goal of delivering proper feedback. In addition, we give a formulation, and if possible a proof, of the properties they satisfy. The formalisation makes the concepts precise, which enables us to reason about them. The feedback services use exercise descriptions for their instances on domains such as logic, algebra, and linear algebra. The exercise descriptions contain the strategy for solving a particular exercise, a predicate that determines whether or not an exercise is solved, etc. The desired properties of our feedback services

H. Kirchner, C. Muñoz (Eds.): First International Workshop on
Strategies in Rewriting, Proving, and Programming (IWS 2010)
EPTCS 44, 2010, pp. 21–34, doi:10.4204/EPTCS.44.2

lead to requirements for the exercise descriptions. We will formulate these requirements, and show how our formalisation helps in verifying that the requirements are satisfied for a particular exercise.

This paper makes the following contributions:

- We give a formal definition of the concepts we use in our feedback framework.
- We formulate the properties we want our feedback services to satisfy.
- We formulate the requirements that an exercise description should fulfil, and we show these requirements are satisfied by an example exercise description.

This paper is organised as follows. Section 2 introduces the fundamental concepts of our framework, such as rewrite rules and strategies. Section 3 lists the services we offer, and formulates the properties that we want our services to satisfy. Section 4 states a number of exercise-specific properties that an exercise description should have. Section 5 gives related work, and Section 6 concludes.

2 Strategy Language

This section gives a formal definition of our rewrite-strategy language. We start the introduction of our rewrite-strategy language for specifying exercises with an example. Consider the problem of rewriting the expression $(a^3 \cdot a^4)^2$ as a power of a, using the following three rewrite rules:

$$\begin{array}{ll} a^x \cdot a^y = a^{x+y} & [\textsc{AddExp}] \\ (a^x)^y = a^{x \cdot y} & [\textsc{MulExp}] \\ (a \cdot b)^x = a^x \cdot b^x & [\textsc{DistExp}] \end{array}$$

A possible strategy to solve the given power expression is to apply these rules bottom-up, until none of the rules can be applied anymore. Applying this strategy to the given expression results in the following derivation: $(a^3 \cdot a^4)^2 \overset{\textsc{AddExp}}{\Longrightarrow} (a^7)^2 \overset{\textsc{MulExp}}{\Longrightarrow} a^{14}$. A strategy captures expert knowledge, in this case how to solve an exercise involving powers. In addition to a strategy, an exercise description consists amongst others of the rules that can be applied to a term, the form of the term that is shown to the user (the exercise from the viewpoint of the user), a predicate for determining whether or not an exercise is solved.

In the remainder of this section we give a formal definition of our strategy language and related concepts.

2.1 Strategy Language Definition

Definition 2.1. *Let μ be the fixed-point combinator $\mu\, f = f\ (\mu\, f)$. A strategy is an element of the language of the following grammar:*

$$\begin{array}{lcl} \sigma & ::= & a \mid \sigma \mathbin{<\!\star\!>} \sigma \mid \sigma \mathbin{<\!|\!>} \sigma \mid \gamma \mid \delta \mid \ell\, \sigma \mid \mu\, f \\ a & ::= & r \mid \sim\!\sigma \end{array}$$

The components of the grammar for σ are called strategy combinators *[8]. Two (sub)strategies can be combined into a strategy using the sequence ($<\!\star\!>$) or choice ($<\!|\!>$) combinator, with γ (always succeeds) and δ (always fails) as unit elements, respectively. A strategy can be tagged with a label (ℓ). The μ combinator returns the fixed-point of a function that takes a strategy as argument and returns a strategy. The non-terminal symbol a is either a rewrite rule r or an applicability check $\sim\!\sigma$ parametrised over a strategy.*

This definition corresponds to the definition of a context-free grammar (CFG), extended with fixed-points, with an alphabet consisting of rewrite rules and applicability checks. We distinguish two kinds of rules: *minor* rules, also called administrative rules, and *major* (normal) rewrite rules. Minor rules are used to perform administrative tasks, such as moving down into a term, updating an environment, or automatically simplifying a term, such as replacing $x+0$ by x. A major rule may be turned into a minor rule to decrease the granularity of intermediate steps (e.g., rewriting $3+4$ to 7 in our running example), or increase the difficulty of an exercise. The function *isMinor* is used to determine whether or not a rewrite rule is minor. When tracking a student working on an exercise we maintain an environment, for example for storing extra information. Major rules typically are the rules a user applies, such as the three rules for manipulating powers given above. The example derivation given above only shows the major rules. The minor rules which move the focus into a term, for example for moving from $(a^3 \cdot a^4)^2$ to $a^3 \cdot a^4$ to apply rule ADDEXP, are not shown in the derivation. The author determines whether or not a rule is major or minor. It is advisable to make only rules which the user can apply major, since major rules will be shown to the user in derivations, and be given as hints. For example, if the focus in the editor cannot be set by the user, it is unwise to make a rule that changes the focus in a term a major rule.

The main purpose of our feedback framework is to track student behaviour, and to automatically calculate feedback based on the strategy and the current term. For this purpose we have to find out *where* in a strategy an error is made, for example. Error messages or hints depend on the position in the strategy at which a student has arrived. To connect an error message to a particular position in a strategy we *label* a position in the strategy. A labeled strategy can be transformed to a non-labeled strategy with additional minor rules, namely ENTER and LEAVE. The ENTER and LEAVE minor rules update the environment in order to keep track where we are in a strategy.

The *language* of a strategy is the set of sequences of rewrite rules and applicability checks returned by function L:

$$\begin{aligned}
L\,(a) &= \{a\} \\
L\,(\sigma_1 \mathbin{<\!\star\!>} \sigma_2) &= \{xy \mid x \in L\,(\sigma_1), y \in L\,(\sigma_2)\} \\
L\,(\sigma_1 \mathbin{<|>} \sigma_2) &= L\,(\sigma_1) \cup L\,(\sigma_2) \\
L\,(\gamma) &= \{\varepsilon\} \\
L\,(\delta) &= \emptyset \\
L\,(\ell\,\sigma) &= L\,(\sigma) \\
L\,(\mu\,f) &= L\,(f\,(\mu\,f))
\end{aligned}$$

For example, using Haskell notation for representing lists and omitting minor rules, the list $[\text{ADDEXP}, \text{MULEXP}]$ is an element of the language of the strategy for solving power exercises. We will give a more detailed example in Section 2.3.

2.2 Derived Combinators

On top of the primitive strategy combinators given in the previous subsection, we define a set of derived strategy combinators. These derived strategy combinators are very useful for formulating rewrite strategies for exercises. They are be built on top of the basic concepts.

An important combinator is the left-biased choice ($\triangleright$), which can be defined as follows:

$$\sigma_1 \triangleright \sigma_2 = \sigma_1 \mathbin{<|>} (\sim\sigma_1 \mathbin{<\!\star\!>} \sigma_2)$$

The strategy σ_2 is only considered when σ_1 is not applicable. Other combinators, such as *try*, *repeat*, and *option*, are similar to the well-known EBNF (extended Backus Naur form) constructs.

$$\begin{aligned} \mathit{option}\ \sigma &= \sigma \mathbin{<|>} \gamma \\ \mathit{try}\ \sigma &= \sigma \mathbin{|>} \gamma \\ \mathit{repeat}\ \sigma &= \mu\ (\lambda x\ .\ \mathit{try}\ (\sigma \mathbin{<*>} x)) \end{aligned}$$

The *option* strategy combinator applies a strategy optionally. The *try* combinator applies a strategy when it is applicable. The *repeat* combinator tries to apply a strategy as many times as possible.

2.3 Navigation

Besides the derived combinators from the previous subsection, we add a set of traversal combinators to our strategy language. Traversal combinators traverse a term, and for example perform rewrite rules or strategies *somewhere* or *bottomUp*. We use a number of administrative rules for navigating through the abstract syntax tree (AST) of an expression: UP, DOWN, LEFT, and RIGHT. The minor rule DOWN takes a function as argument, which decides which child to select based on the environment. Using DOWN we construct a minor rule that selects all children using the choice combinator: DOWNS. Navigation is implemented by means of a zipper [10], which is an efficient data structure to define and move a focus in an expression. The zipper can be seen as a combination of an expression and its context. An alternative way to navigate is to use position information of (sub)expressions. An implementation using this approach uses a list of integers denoting a path from the top of the expression to the subexpression in focus. This approach is not as efficient and type-safe as a zipper, since the AST needs to be traversed to retrieve the subexpression in focus, and since it is possible to specify paths that do not correspond to a position in the tree.

Many traversal combinators use the *once* combinator:

$$\mathit{once}\ \sigma = \textsc{Downs} \mathbin{<*>} \sigma \mathbin{<*>} \textsc{Up}$$

The *once* combinator takes a strategy as argument, and applies it to a direct child of the expression currently in focus. After applying *once* the focus is again at the top-level expression. The *once* combinator returns all possible ways, by means of the choice combinator introduced by DOWNS, in which a strategy can be applied once to a direct child of an expression. So an application of a strategy constructed with the *once* combinator may have more than one result, depending on whether or not strategy σ is applicable to the children.

The traversal combinator *somewhere* applies a strategy to a single subexpression (including the expression itself).

$$\mathit{somewhere}\ \sigma = \mu\ (\lambda x\ .\ \sigma \mathbin{<|>} \mathit{once}\ x)$$

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

$$\begin{aligned} \mathit{bottomUp}\ \sigma &= \mu\ (\lambda x\ .\ \mathit{once}\ x \mathbin{|>} \sigma) \\ \mathit{topDown}\ \sigma &= \mu\ (\lambda x\ .\ \sigma \mathbin{|>} \mathit{once}\ x) \end{aligned}$$

These combinators search for a suitable location to apply an argument strategy in a bottom-up or top-down fashion, without imposing an order in which the children are visited. These combinators do not apply their argument strategy exhaustively, instead, the argument strategy is applied only once.

Navigation operators navigate through the abstract syntax of the domain on which the rewrite rules are specified. The example rules we presented at the beginning of this section work on algebraic expressions containing powers. We give a formal definition of the power domain, and the zipper we use for navigation on this domain.

$$\begin{array}{lcl} e & ::= & v \mid e^n \mid e \cdot e \\ \varphi & ::= & [\![e]\!] \mid \varphi^n \mid \varphi \cdot e \mid e \cdot \varphi \end{array}$$

Here v stands for a variable and n for an integer number. In the grammar for the zipper, the expression between double square brackets is the expression in focus. The focus can appear inside a power φ^n, or move left or right into a product ($\varphi \cdot e$, or $e \cdot \varphi$). The minor rules for navigation are defined by analysing the various forms of the zipper. For example, the UP rule is defined as follows:

$$\begin{array}{lll} \text{UP } \varphi = \textbf{case } \varphi \textbf{ of } & [\![e]\!]^i & \to [\![e^i]\!] \\ & [\![e_1]\!] \cdot e_2 & \to [\![e_1 \cdot e_2]\!] \\ & e_1 \cdot [\![e_2]\!] & \to [\![e_1 \cdot e_2]\!] \\ & \varphi^n & \to (\text{UP } \varphi)^n \\ & \varphi \cdot e & \to \text{UP } \varphi \cdot e \\ & e \cdot \varphi & \to e \cdot \text{UP } \varphi \end{array}$$

We return to the example we gave at the beginning of this section. Now that we have a formal definition of strategies, we can express the informal strategy in terms of the derived strategy combinators.

$$\mathit{writeAsPowerOf} = \ell \; (\mathit{repeat} \; (\mathit{bottomUp} \; (\text{ADDEXP} \mathbin{<|>} \text{MULEXP} \mathbin{<|>} \text{DISTEXP})))$$

Applying the *writeAsPowerOf* strategy to the expression $(a^3 \cdot a^4)^2$ gives the following derivation:

$$\begin{array}{l} [\![(a^3 \cdot a^4)^2]\!] \overset{(\text{ENTER } \ell)}{\Longrightarrow} [\![(a^3 \cdot a^4)^2]\!] \overset{\text{DOWN}}{\Longrightarrow} [\![a^3 \cdot a^4]\!]^2 \overset{\text{APPCHECK}}{\Longrightarrow} [\![a^3 \cdot a^4]\!]^2 \overset{\text{ADDEXP}}{\Longrightarrow} \\ [\![a^7]\!]^2 \overset{\text{UP}}{\Longrightarrow} [\![(a^7)^2]\!] \overset{\text{APPCHECK}}{\Longrightarrow} [\![(a^7)^2]\!] \overset{\text{MULEXP}}{\Longrightarrow} [\![a^{14}]\!] \overset{\text{APPCHECK}}{\Longrightarrow} [\![a^{14}]\!] \overset{(\text{LEAVE } \ell)}{\Longrightarrow} [\![a^{14}]\!] \end{array}$$

The APPCHECK (applicability check) is introduced by the *repeat* and *bottomUp* combinator. The navigation rules do not work directly on an expression, but on the zipper containing the expression (φ). As a consequence, if a strategy uses a traversal combinator it is only applicable to an expression in a context. (In the example derivation above we omit the context from the rewrite steps, and only show the expression and the focus.) To keep the definition of rewrite rules simple and clean, we have a function that lifts a rewrite rule to operate on an expression in a context. The definition of this function is omitted.

2.4 Semantics

At the start of this section we defined the language of a strategy. The language contains lists of rewrite rules and applicability conditions. In this subsection we define how a strategy transforms an expression. We use the following terminology in the definition of the semantics:

Expression. We only consider expressions generated by a grammar.

Environment. An environment stores additional information at intermediate steps, such as auxiliary results. An environment is a set of key/value pairs, which can be added, removed, consulted, and updated.

Rewrite rule. A rewrite rule r is a binary relation on the product of an environment and an expression: $(\Gamma_1 \times e_1) \overset{r}{\rightsquigarrow} (\Gamma_2 \times e_2)$. A rewrite rule is tagged with a boolean indicating whether or not it is a minor rule. If the rewrite rule does not change the environment we use the notation $e_1 \overset{r}{\rightsquigarrow} e_2$.

A lifted rewrite rule is a binary relation on the product of an environment and a zipper (an expression in focus together with its context): $(\Gamma_1 \times \varphi_1) \overset{r}{\rightsquigarrow} (\Gamma_2 \times \varphi_2)$.

State. The state of an exercise is modelled as the product of an environment, a zipper, and a strategy.

To check if we have reached an end state we want to determine whether or not the empty sentence is a member of the language of a strategy. For this purpose we define a function *empty*, similar to the function *empty* defined for CFGs. We first define an auxiliary function that returns all sentences that consist of minor rules only:

Definition 2.2. *Function minorSentences returns all sentences in the language of a strategy that consist of minor rules only, including the empty sentence ε:*

$$\mathit{minorSentences}\ (\sigma) = \{\, r_1\ \ldots\ r_n \mid r_1\ \ldots\ r_n\ \in\ L\ (\sigma), \forall\, i\ \in\ 1\ \ldots\ n\ .\ \mathit{isMinor}\ (r_i)\,\}$$

We need this function to determine if there are any trailing minor rules after the last major rewrite rule. We define the *empty* function in terms of the *minorSentences* function:

Definition 2.3. *Function empty checks whether or not the language of a strategy contains the empty sentence ε, or a sentence consisting of minor rules only:*

$$\mathit{empty}\ (\sigma) = \mathit{minorSentences}\ (\sigma)\ \neq\ \emptyset$$

Both functions are easily lifted to take a state as argument.

The smallest action that can be performed with a strategy is a *step*: the application of a rewrite rule. Before we define a step relation that performs a state transition, we define a relation that splits a strategy into its first rule or applicability check and the remaining strategy.

Definition 2.4. *The relation $\mapsto$ splits a strategy into a rule or an applicability check and the remaining strategy: $\sigma_1 \mapsto a \mathbin{<\!\star\!>} \sigma_2$.*

$$a \mapsto a \mathbin{<\!\star\!>} \gamma \qquad \frac{\sigma_1 \mapsto a \mathbin{<\!\star\!>} \sigma_3}{\sigma_1 \mathbin{<\!\star\!>} \sigma_2 \mapsto a \mathbin{<\!\star\!>} (\sigma_3 \mathbin{<\!\star\!>} \sigma_2)} \qquad \frac{\varepsilon\ \in\ L\ (\sigma_1) \qquad \sigma_2 \mapsto a \mathbin{<\!\star\!>} \sigma_3}{\sigma_1 \mathbin{<\!\star\!>} \sigma_2 \mapsto a \mathbin{<\!\star\!>} \sigma_3}$$

$$\frac{\sigma_1 \mapsto a \mathbin{<\!\star\!>} \sigma_3}{\sigma_1 \mathbin{<\!\diamond\!>} \sigma_2 \to a \mathbin{<\!\star\!>} \sigma_3} \qquad \frac{\sigma_2 \mapsto a \mathbin{<\!\star\!>} \sigma_3}{\sigma_1 \mathbin{<\!\diamond\!>} \sigma_2 \to a \mathbin{<\!\star\!>} \sigma_3} \qquad \frac{f\ (\mu\, f) \mapsto a \mathbin{<\!\star\!>} \sigma}{\mu\, f \mapsto a \mathbin{<\!\star\!>} \sigma}$$

$$\ell\ \sigma \mapsto \textsc{Enter}\ \ell \mathbin{<\!\star\!>} (\sigma \mathbin{<\!\star\!>} \textsc{Leave}\ \ell)$$

Definition 2.5. *The step operator $\xrightarrow{r}$ denotes the relation between the current state S_1 and a new state S_2 (obtained by applying the rewrite rule r).*

$$\frac{\sigma_1 \mapsto r \mathbin{<\!\star\!>} \sigma_2 \qquad (\Gamma_1 \times \varphi_1) \overset{r}{\rightsquigarrow} (\Gamma_2 \times \varphi_2)}{(\Gamma_1 \times \varphi_1 \times \sigma_1) \xrightarrow{r} (\Gamma_2 \times \varphi_2 \times \sigma_2)} \qquad \frac{\sigma_1 \mapsto \sim\sigma_2 \mathbin{<\!\star\!>} \sigma_3 \qquad \mathit{run}\ (\Gamma_1 \times \varphi_1 \times \sigma_2) = \emptyset}{(\Gamma_1 \times \varphi_1 \times \sigma_3) \xrightarrow{r} (\Gamma_2 \times \varphi_2 \times \sigma_4)} \qquad \frac{}{(\Gamma_1 \times \varphi_1 \times \sigma_1) \xrightarrow{r} (\Gamma_2 \times \varphi_2 \times \sigma_4)}$$

The *run* function used in the last rule applies a strategy to a term in a context; its definition is given below.

The step relation $\xrightarrow{r}$ ignores whether or not a rule is minor, and it deals with minor and major rewrite rules in the same way. Many of the feedback services we have defined ignore minor rewrite rules, since we do not want to show such administrative steps to a user. We define a relation similar to step, which ignores minor rewrite rules.

Definition 2.6. *The big step operator $\overset{r}{\twoheadrightarrow}$ denotes the relation between a state S_1 and a new state S_2. The new state is obtained by possibly applying minor rules, followed by the application of a single major rewrite rule r. If r is the last major rule then trailing minor rules, if any, are applied as well.*

$$\frac{S_1 \overset{r_1}{\rightarrow} S_2 \quad \mathit{isMinor}\ (r_1) \quad S_2 \overset{r_2}{\twoheadrightarrow} S_3}{S_1 \overset{r_2}{\twoheadrightarrow} S_3}$$

$$\frac{S_1 \overset{r}{\rightarrow} S_2 \quad \neg\ \mathit{isMinor}\ (r) \quad \mathit{minorSentences}\ (S_2) = \emptyset}{S_1 \overset{r}{\twoheadrightarrow} S_2}$$

$$\frac{S_1 \overset{r}{\rightarrow} S_2 \quad \neg\ \mathit{isMinor}\ (r) \quad \vec{m} \in \mathit{minorSentences}\ (S_2) \quad S_2 \overset{\vec{m}}{\rightarrow} S_3}{S_1 \overset{r}{\twoheadrightarrow} S_3}$$

where $\vec{m}$ is a sequence of minor rules and $\overset{\vec{m}}{\rightarrow}$ is the sequential application thereof.

It is important, when performing a big step, that trailing minor rules are applied. The application of trailing minor rules ensure that exhaustively applying the step or big step operator on a term, will end up in the same end state(s). This enables us to keep the generated feedback consistent.

Definition 2.7. *The run function is the closure of $\overset{r}{\twoheadrightarrow}$, and generates all possible end states from a begin state S_0. An end state contains the empty strategy: γ.*

$$\mathit{run}\ (S_0) = \{\, (\Gamma \times \varphi \times \gamma) \mid S_0 \overset{\star}{\twoheadrightarrow} (\Gamma \times \varphi \times \gamma) \,\}$$

where $\overset{\star}{\twoheadrightarrow}$ is the transitive closure of $\twoheadrightarrow$.

2.5 Non-determinism

Almost all exercises can be solved in several, correct ways. For example, consider the expression $(a^3 \cdot a^4)^2$ again, and suppose it should be solved with the strategy *writeAsPowerOf′* which is obtained by replacing *bottomUp* by *somewhere* in the strategy *writeAsPowerOf*.

$$\mathit{writeAsPowerOf'} = \ell\ (\mathit{repeat}\ (\mathit{somewhere}\ (\textsc{AddExp} \mathbin{<|>} \textsc{MulExp} \mathbin{<|>} \textsc{DistExp})))$$

One of the questions we want to be able to answer is: what is the next step in order to solve the exercise? This type of feedback is given by the *onefirst* feedback service, which we will describe in more detail in the next section. In the above example there are two possibilities: DISTEXP can be applied to the entire expression, and ADDEXP is applicable to the subexpression $a^3 \cdot a^4$. Both steps are correct, but which do we choose? Making a random choice would make our feedback framework non-deterministic. This problem may show up whenever we use the choice combinator, which may also be introduced by other combinators (such as DOWNS), to combine various solution strategies. Applying a strategy that uses the choice combinator to an expression may result in multiple answers.

To prevent non-deterministic behaviour we introduce a rule ordering: a total order, denoted by $<$, on rules. For now, the ordering is only on the rules themselves, but one can imagine taking the environment into account in the rule ordering. An example rule ordering for the power domain is: ADDEXP $<$ MULEXP $<$ DISTEXP. In this case, the first step in our example is ADDEXP.

2.6 Restrictions

To use strategies to track student behaviour and give feedback, we impose some restrictions on the form of strategies. These restrictions are similar to some of the restrictions imposed on context-free grammars to be able to use them for parsing.

Left-recursion. A context-free grammar is left-recursive if it contains a nonterminal that can be rewritten in one or more steps using the productions of the grammar to a sequence of symbols that starts with the same nonterminal. The same definition applies to strategies. For example, the following strategy is left-recursive:

$$\mathit{leftRecur} = \mu\ (\lambda x\ .\ x \mathbin{<\!\star\!>} \textsc{AddExp})$$

The left-recursion is obvious in this strategy, since x is in the leftmost position in the body of the abstraction. Left-recursion is not always this easy to spot. Strategies with leading minor rules may or may not be left-recursive. Strictly speaking, these strategies are not left-recursive because the strategy grammar does not differentiate between minor and major rules. However, our semantics make it that these strategies sometimes display left-recursive behaviour. For example:

$$\mathit{leftRecur}' = \mu\ (\lambda x\ .\ \textsc{Down} \mathbin{<\!\star\!>} x \mathbin{<\!\star\!>} \textsc{AddExp})$$

Here, applying the minor rule DOWN repeatedly will eventually cause the strategy to reach the leaf of an expression tree, and stop. Hence this strategy is not left-recursive. However, this is a property of DOWN that is not shared by all other minor rules. If we use a minor rule that increases a counter in the environment, an action that will always succeed, the strategy is left-recursive.

Top-down recursive parsing using a left-recursive context-free grammar is difficult. We have chosen for top-down recursive parsing because we need to parse a derivation incrementally. Other parsing schemes, such as bottom-up parsing, also have their difficulties. A grammar represented in parser combinators [11] is not allowed to be left-recursive. Similarly, for a strategy to be used in our framework, it should not be left-recursive. Trying to determine the next possible symbol(s) of a left-recursive strategy by means of the split operator will get stuck in a loop.

Left-recursive strategies are not the only source of non-terminating strategy calculations. The fact that our strategy language has a fixed-point combinator implies that we are vulnerable to non-termination. The implementation of our strategy language has been augmented with 'time-outs' that will stop the execution and report an error message.

Left-factorisation. Left-factoring is a grammar transformation that is useful when two productions for the same nonterminal start with the same sequence of terminal and/or nonterminal symbols. This transformation factors out the common part of such productions. In a strategy, the equivalent transformation factors out common sequences of rewrite rules from sub-strategies separated by the choice combinator.

A strategy that contains left-factors may be problematic. Consider the following, somewhat contrived, strategy:

$$\mathit{leftStrat} = \ell_1\ (\textsc{AddExp} \mathbin{<\!\star\!>} \textsc{MulExp}) \mathbin{<|>} \ell_2\ (\textsc{AddExp} \mathbin{<\!\star\!>} \textsc{DistExp})$$

The two sub-strategies labeled with ℓ_1 and ℓ_2 have a left-factor: the rewrite rule ADDEXP. After the application of ADDEXP, we have to decide which sub-strategy to follow. Either we follow sub-strategy

ℓ_1, or we follow sub-strategy ℓ_2. Committing to a choice after seeing an application ADDEXP is unfortunate, since it will force the student to follow the same sub-strategy. For example, if ℓ_1 is chosen after an application of ADDEXP has been submitted, and a student subsequently takes the DISTEXP step, we erroneously report that the step does not follow the strategy. Therefore, left-factorising a strategy is desirable, since we do not want to commit early to a particular sub-strategy. The example strategy can be left-factored as follows:

$$\mathit{leftStrat}' = \text{ADDEXP} \mathbin{<\!\!*\!\!>} (\text{MULEXP} \mathbin{<|>} \text{DISTEXP})$$

It is clear how to left-factor (major) rewrite rules, but how should we deal with labels, or minor rules in general? In the remainder of this section we focus on how to deal with labels. Pushing the labels inside the choice combinator,

$$\mathit{leftStrat}' = \text{ADDEXP} \mathbin{<\!\!*\!\!>} (\ell_1\ \text{MULEXP} \mathbin{<|>} \ell_2\ \text{DISTEXP})$$

or making a choice between the two labels probably breaks the relation between the label and the strategy. Recall that labels are used to mark positions in a strategy, which is connected to a feedback text.

A different way to deal with left-factors uses backtracking. With backtracking we remember the start position of the left-factor in the strategy. In the case that the chosen sub-strategy fails we roll back and continue with a different sub-strategy. However, using backtracking it is possible to guide a student to a dead end (a sub-strategy that fails). This violates our goals of giving proper and relevant feedback.

Our solution is to detect and report left-factors. The detection of left-factors in a strategy is relatively straightforward. The detection and notification enables strategy developers to construct strategies without left-factors.

3 Services

Our feedback services offer a wide variety of feedback functionality, so that learning environments using our services can provide different kinds of feedback. A learning environment can amongst others ask for the following kinds of feedback: 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? Has the student made a common error? What does a worked-out solution look like? This section formalises the corresponding services.

3.1 Exercises

Most of the feedback services calculate feedback based on a strategy, but sometimes there are other components that play a role in our services. All components necessary for our feedback services are grouped together in an *exercise*. An exercise contains all domain-specific (and exercise-specific) functionality, together with an exercise code for identification. The most important component of an exercise is its strategy. Additional rewrite rules can be added to an exercise to help detect possible detours. We not only specify proper rewrite rules, but also *buggy rules*. A buggy rule captures a common misconception. If we detect an application of a buggy rule, we report this to the user. We also need predicates for checking whether or not an expression is a suitable starting expression that can be solved by the strategy, and whether or not an expression is in solved form. For diagnosing intermediate answers, we need an equivalence relation to compare a submission with a preceding expression, and a similarity relation, which

is possibly more liberal than syntactic equality. Although not of primary importance, it can be convenient to have a randomised expression generator for the exercise. The last component of an exercise is a function that returns the order of rules.

Definition 3.1. *An exercise consists of an identification code, a strategy σ, a rule set $\{r_1 \ldots r_n\}$, a buggy rule set, an equivalence relation $\equiv$, a similarity relation $\approx$, a predicate isSuitable, a predicate isReady, an optional expression generator, and a rule ordering function $<$.*

An example of an exercise is the exercise for our running example: calculating with powers.

$$\begin{aligned} \mathit{powerExercise} = (&\texttt{powerExercise} \\ &, \ell\,(\mathit{repeat}\,(\mathit{bottomUp}\,(\textsc{AddExp} \mathbin{<|>} \textsc{MulExp} \mathbin{<|>} \textsc{DistExp}))) \\ &, \{a^x = \tfrac{1}{a^{-x}} \quad [\textsc{ReciExp}]\} \\ &, \{a^x \cdot a^y = a^{x \cdot y} \quad [\textsc{BugAddExp}]\} \\ &, \mathit{eqPower}, \mathit{simPower}, \mathit{suitablePower}, \mathit{readyPower} \\ &, \textsc{DistExp} < \textsc{MulExp} < \textsc{AddExp}) \end{aligned}$$

Here, *eqPower*, *simPower*, *suitablePower*, and *readyPower* are defined as follows, where $\simeq$ stands for syntactic equality:

$$\begin{aligned} \mathit{normPower}\ v &= v \\ \mathit{normPower}\ (e_1 \cdot e_2) &= \mathit{simplifyPower}\ (\mathit{normPower}\ e_1 \cdot \mathit{normPower}\ e_2) \\ \mathit{normPower}\ (e^n) &= \mathit{simplifyPower}\ ((\mathit{normPower}\ e)^n) \\ \mathit{simplifyPower}\ ((a^x)^y) &= a^{x \cdot y} \\ \mathit{simplifyPower}\ (a^x \cdot a^y) &= a^{x+y} \\ \mathit{simplifyPower}\ ((a \cdot b)^x) &= a^x \cdot b^x \\ \mathit{simplifyPower}\ e &= e \\ \mathit{eqPower}\ e_1\ e_2 &= \mathit{normPower}\ e_1 \simeq \mathit{normPower}\ e_2 \\ \mathit{simPower}\ e_1\ e_2 &= e_1 \simeq e_2 \\ \mathit{suitablePower}\ e &= \mathit{normPower}\ e \not\simeq e \\ \mathit{readyPower}\ e &= \mathit{normPower}\ e \simeq e \end{aligned}$$

We have simplified the implementation of *normPower* by ignoring the fact that $\cdot$ is associative. In this example, we have chosen to use normalisation as a base to define the exercise-specific functions, such as $\equiv$. However, this is not necessarily the case for other domains.

3.2 Formalised Services

We define the set of services we offer to learning environments in terms of the relations defined in the previous section.

allfirsts. The *allfirsts* service returns all next steps that are allowed by a strategy in a particular state:

$$\mathit{allfirsts}\ S_0 = \{(S, r) \mid S_0 \xrightarrow{r} S\}$$

Consider the following state S:

$$\begin{aligned} S = (\Gamma, [\![(a^3 \cdot a^4)^2]\!], ((&\mathit{somewhere}\ \textsc{AddExp}) \mathbin{<\!\star\!>} \textsc{MulExp}) \mathbin{<|>} \\ &(\textsc{DistExp} \mathbin{<\!\star\!>} \mathit{repeat}\ \textsc{MulExp} \mathbin{<\!\star\!>} \textsc{AddExp})) \end{aligned}$$

An *allfirsts S* service call gives the following result. The UP minor rule is introduced by the somewhere combinator to get the focus back to its original place.

$$\{((\Gamma,([\![a^7]\!])^2, \quad \text{UP} \mathbin{<\!\!*\!\!>} \text{MULEXP}), \text{ADDEXP}) \\ ,((\Gamma,[\![(a^3)^2\cdot(a^4)^2]\!], \textit{repeat}\ \text{MULEXP} \mathbin{<\!\!*\!\!>} \text{ADDEXP}), \text{DISTEXP})\}$$

onefirst. The *onefirst* service returns a single possible next step that follows the strategy. This service uses the *allfirsts* service and the rule ordering.

$$\textit{onefirst}\ S_0 = (S,r)\ \textbf{where}\ (S,r)\ \in\ \textit{allfirsts}\ S_0\ \wedge\ \forall\,(S_i,r_i)\ \in\ \textit{allfirsts}\ S_0\ .\ r\ \leqslant\ r_i$$

Performing a *onefirst* service call on the state *S* from the previous paragraph, taking into account the rule ordering from subsection 2.5, gives following result:

$$((\Gamma,([\![a^7]\!])^2, \text{UP} \mathbin{<\!\!*\!\!>} \text{MULEXP}), \text{ADDEXP})$$

derivation. The *derivation* service returns a worked-out solution of an exercise starting with the current expression.

$$\textit{derivation}\ S_0 = (S_1,r_1)\ (S_2,r_2)\ \ldots\ (S_n,r_n)\ \textbf{where}\ \textit{empty}\ (S_n)\ \wedge \\ \forall\, i\ \in\ 1\ \ldots\ n\ .\ (S_{i-1},r_i) = \textit{onefirst}\ S_i$$

ready. The *ready* service checks if the expression in a state is in a form accepted as a final answer. The *ready* service is an interface to the *isReady* predicate defined in an exercise.

stepsremaining. The *stepsremaining* service computes how many steps remain to be done according to the strategy. This is achieved by calculating the length of a derivation.

apply. The *apply* service applies a rule to an expression at a particular location, regardless of the strategy. The location is represented as a list of integers, where each integer *n* represents the number of steps to the right after a step downwards in a subexpression (the *n*th child). Starting at the root of an expression we can assign every subexpression a unique location.

The function *setFocus* translates a location to a sequence of minor rules that puts the focus at a particular subexpression.

$$\textit{setFocus}\ S_0\ [\,] \quad = S_0 \\ \textit{setFocus}\ S_0\ (n:ns) = \textit{setFocus}\ S_1\ ns\ \textbf{where}\ S_0 \overset{(\text{DOWN}\,(\textit{const}\ n))}{\rightarrow} S_1$$

The function *focusToRoot* sets the focus to the root of an expression. We omit the definition of this function. The *apply* service is defined as follows:

$$\textit{apply}\ r\ \textit{loc}\ S_0 = S_1\ \textbf{where}\ \textit{setFocus}\ \textit{loc}\ (\textit{focusToRoot}\ S_0) \xrightarrow{r} S_1$$

For example, *apply* MULEXP $[1]$ $(\Gamma,[\![(a^3)^2\cdot(a^4)^2]\!],\sigma)$ gives $(\Gamma,((a^3)^2\cdot[\![a^8]\!]),\sigma)$.

applicable. The *applicable* service takes an expression and a location in this expression, and returns all major rules that can be applied at this location, independent of the strategy. Let *R* be the union of the rules in the strategy and the exercise rule set, then *applicable* is defined as follows:

$$\textit{applicable}\ \textit{loc}\ S_0 = \{\,r \mid r\ \in\ R, S_1 \xrightarrow{r} S_2\,\}\ \textbf{where}\ S_1 = \textit{setFocus}\ \textit{loc}\ (\textit{focusToRoot}\ S_0)$$

generate. The *generate* service takes an exercise code and a difficulty level (optional), and returns an initial state with a freshly generated expression.

diagnose. The *diagnose* service diagnoses an expression submitted by a student. Possible diagnoses are:

- *NotEq*: the current and submitted expression are not equivalent, i.e. something is wrong,
- *Buggy*: a common misconception has been detected,
- *Similar*: the expression is similar to the last expression in the derivation,
- *Expected*: the submitted expression is expected by the strategy,
- *Detour*: the submitted expression was not expected by the strategy, but the applied rule was detected,
- *Correct*: the submitted expression is correct, but we cannot determine which rule was applied.

The *diagnose* service is defined as follows, where the *unFocus* function converts a focussed expression to a normal expression:

$$
\begin{array}{l}
\mathit{diagnose}\ (\Gamma, \varphi, \sigma)\ \mathit{new} = \\
\quad \textbf{if}\ \mathit{unFocus}\ \varphi \not\equiv \mathit{new} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then} \\
\qquad \textbf{if}\ \exists\, b \in \mathit{buggyRuleSet} \,.\, \exists\, e \,.\, \mathit{unFocus}\ \varphi \overset{b}{\rightsquigarrow} e \wedge e \equiv \mathit{new} \qquad\qquad \textbf{then}\ \mathit{Buggy} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{NotEq} \quad \textbf{else} \\
\quad \textbf{if}\ \mathit{unFocus}\ \varphi \approx \mathit{new} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{Similar} \quad \textbf{else} \\
\quad \textbf{if}\ \mathit{new} \in \mathit{allfirsts}\ (\Gamma, \varphi, \sigma) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{Expected}\ \textbf{else} \\
\quad \textbf{if}\ \exists\, r \in \mathit{ruleSet} \,.\, \exists\, e \,.\, \mathit{unFocus}\ \varphi \overset{r}{\rightsquigarrow} e \wedge e \equiv \mathit{new} \qquad\qquad\qquad\quad \textbf{then}\ \mathit{Detour} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{Correct}
\end{array}
$$

We conclude this section with a soundness result. We give a theorem connecting the language concept, the function L, to the services built on top of strategies defined in this section. We start with the introduction of a lemma that simplifies the proof of the theorem.

Lemma 3.2. *A major language $\mathscr{L}$ is the language of a strategy without occurrences of minor rules:*

$$\mathscr{L}\,(\sigma) = \{\,[a \mid a \leftarrow as, \neg\, \mathit{isMinor}\ a] \mid as \in L\,(\sigma)\,\}$$

Let F be the set of all splits given by the split relation for σ: $\{\,\sigma_i \mid \sigma \mapsto \sigma_i\,\}$. Then the major language of σ without the empty sentence is equal to the union of major languages of the elements in F:

$$\mathscr{L}\,(\sigma) \setminus \{\,\varepsilon\,\} \equiv \bigcup_{\sigma_f \in F} \mathscr{L}\,(\sigma_f)$$

Proof. By case analysis on the structure of σ. □

Theorem 3.3. *Let S_0 be a state $(\Gamma \times \varphi \times \sigma)$ and $\vec{r}$ be a sequence of rules $[r_1 \ldots r_n]$ such that derivation $(S_0) = (S_1, r_1) \ldots (S_n, r_n)$. Then $\vec{r} \in \mathscr{L}\,(\sigma)$.*

Proof. We sketch a proof. Via the definitions of *derivation*, *onefirst*, and *allfirsts* we know that for every element in the derivation sequence $[r_1 \ldots r_n]$ a big step has to be performed. We distinguish two cases in our hypothesis $[r_1 \ldots r_n] \in \mathscr{L}\,(\sigma)$:

$n = 0$: From the statement above we deduce that the strategy σ is either γ or consists of a strategy with minor rules only. From the definition of the major language $\mathscr{L}$ we know $\varepsilon \in \mathscr{L}\,(\sigma)$.

$n > 0$: This case is proven by induction on the length (n) of the sequence $[r_1 \ldots r_n]$ using Lemma 3.2.

□

4 Exercise Properties

The following lemmas express properties that the components in an exercise should have. The lemmas connect the various components of an exercise. The proof of these lemmas for a particular exercise indicates the soundness of the corresponding exercise specification.

Lemma 4.1. *Let S_0 be a state $(\Gamma, [\![e]\!], \sigma)$ and derivation $S_0 = (S_1, r_1)\ \ldots\ (S_n, r_n)$. If e is a suitable start expression, the expression in the last state (S_n) is ready.*

Lemma 4.2. *Let r be a rewrite rule for a particular exercise: $e_1 \overset{r}{\rightsquigarrow} e_2$. Then r is semantics preserving, i.e., $e_1 \equiv e_2$.*

Corollary 4.3. *Let S_0 be a state and derivation $S_0 = (S_1, r_1)\ \ldots\ (S_n, r_n)$. Then all expressions in S_0 up to S_n are in the same equivalence class determined by the exercise's equivalence relation $\equiv$.*

Lemma 4.4. *Let E be the set of expressions generated by the exercise's generator. All expressions $e \in E$ are suitable and not ready: suitable $(e) \wedge \neg$ ready (e).*

Lemma 4.5. *Let S_0 be a state, onefirst $S_0 = (S_1, r_1)$ and allfirsts $S_0 = \{(S_2, r_2)\ \ldots\ (S_n, r_n)\}$. Then $r_1 \equiv min\ \{r_2\ \ldots\ r_n\}$ with respect to the rule ordering.*

The proofs of these lemmas for the exercise for calculating with powers, are rather easy. This is because the rules that are used in the strategy are the same as the rules used to calculate the normal form of a power expression. We provide the proofs for these lemmas in an accompanying technical report [6].

5 Related Work

The strategy language on which our work is based is very similar to languages that are used in parser libraries [15]. Some differences, such as the usage of labels, the focus on intermediate answers, and the concept of minor rules, make the existing libraries unsuitable for generating feedback.

Our language is also similar to strategic programming languages such as Stratego [16] and Elan [4], and tactic languages used in theorem proving. We compare our approach with two more formal approaches to strategies. Kaiser and Lämmel construct a mechanised formal model of Stratego [12]. Our strategy language is different from Stratego in the sense that we, in addition to the final term, also focus on the intermediate rewrite steps. We do not focus on the development of a mechanised model: instead, we give a formal definition of our feedback services that use our strategy language. Tacticals, proof plans and methods [5] 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. Aspinall et al. [1] introduce the tactic language Hitac that can be used to construct hierarchical proofs, so called hiproofs. To evaluate Hitac programs two semantics are given: a big step semantic that captures the intended meaning and a small step semantic that covers the details of the proof. As far as we found, the tactic language is not used to generate feedback, or to recognise proving steps made by a student. Moreover, we provide a set of services that enables learning environments to access our functionality.

6 Conclusions

In this paper we have presented a formal and precise definition of the main concepts that we use to construct semantically rich feedback in learning environments. In addition to a precise definition of our

strategy language we define a step, and a big step relation. These relations give the semantics of the strategy language. Feedback services are an interface to our feedback functionality that can be used by learning environments. These services are expressed in terms of the big step relation.

Our formalisation gives us more confidence in the design choices we have made. Furthermore, we can now validate our current implementation using the properties we state. In the future we want to extend the work in this paper by providing proofs for other domains, such as linear algebra and propositional logic.

References

[1] D. Aspinall, E. Denney, and C. Lüth. A tactic language for hiproofs. In *Intelligent Computer Mathematics*, volume 5144 of *LNCS*, pages 339–354. Springer-Verlag, 2008.

[2] J. Biggs and C. Tang. *Teaching for Quality Learning at University*. Open University Press, 2007.

[3] P. Boon and P. Drijvers. Algebra en applets, leren en onderwijzen (algebra and applets, learning and teaching, in Dutch). `http://www.fi.uu.nl/publicaties/literatuur/6571.pdf`, 2005.

[4] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[5] A. Bundy. The use of explicit plans to guide inductive proofs. In *CADE'88*, pages 111–120, 1988.

[6] A. Gerdes, B. Heeren, and J. Jeuring. Properties of exercise strategies. Technical Report UU-CS-2010-028, Department of Information and Computing Sciences, Utrecht University, 2010.

[7] A. Gerdes, B. Heeren, J. Jeuring, and S. Stuurman. Feedback services for exercise assistants. In Dan Remenyi, editor, *The Proceedings of the 7th European Conference on e-Learning*, pages 402–410. Academic Publishing Limited, 2008.

[8] B. Heeren and J. Jeuring. Recognizing strategies. In Aart Middeldorp, editor, *WRS 2008: Reduction Strategies in Rewriting and Programming, 8th International Workshop*, 2008.

[9] B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 2010.

[10] G. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[11] G. Hutton. Higher-order Functions for Parsing. *JFP*, 2(3):323–343, 1992.

[12] M. Kaiser and R. Lämmel. An Isabelle/HOL-based model of Stratego-like traversal strategies. In *PPDP'09*, pages 93–104. ACM, 2009.

[13] J. Lodder, J. Jeuring, and H. Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.

[14] E. Melis et al. ACTIVEMATH: a generic and adaptive web-based learning environment. *International Journal of Artificial Intelligence in Education*, 12, 2001.

[15] D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In *AFP'96*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[16] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98*, pages 13–26, 1998.