Interleaving Strategies

Bastiaan Heeren[1] and Johan Jeuring[1,2]

[1]School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
{bhr, jje}@ou.nl
[2] Department of Information and Computing Sciences, Universiteit Utrecht

Abstract. Rewrite strategies are used to specify how mathematical exercises are solved in interactive learning environments, and to provide feedback to students solving such exercises. We have developed a generic strategy language with which we can specify rewrite strategies in many (mathematical) domains. Although our strategy language is quite powerful, it lacks an essential component for specifying strategies, namely the *interleaving* of two strategies. Often students have to perform multiple subtasks, but the order in which these tasks are performed is irrelevant, and steps of solutions may be interleaved. We show the need for combinators that support interleaving by means of several examples. We extend our strategy language with different combinators for interleaving, define the semantics of the extension, and show how the interleaving combinators are implemented in the parsing framework we use for recognizing student behavior and providing hints.

Keywords: strategy language, interleaving, feedback

1 Introduction

Strategies specify how a wide range of exercises can be solved incrementally, such as bringing a logic proposition to disjunctive normal form, reducing a matrix, solving a quadratic equation, or calculating with fractions. In previous work [13] we have developed a language for rewrite strategies for exercises and a framework for feedback services built on top of this language, which is now used in intelligent tutoring systems such as MathDox [7], the Digital Mathematics Environment of the Freudenthal Institute [8], and the ActiveMath system [17] to follow student behavior, report applications of buggy rules, give hints, and show worked-out examples.

Rewrite strategies for exercises and the strategy language in which they are formulated should satisfy a number of requirements:

- The strategy language is *generic*: it can be used for any domain in which exercises are solved by incrementally applying rules, such as logic, algebra, programming, etc.
- It is possible to *automatically calculate feedback* given a strategy and actions of a user on an exercise that is solved using the strategy.

– Strategies satisfy the *cognitive fidelity* principle [3]: they reflect textbook descriptions of procedures for solving exercises.
– Strategies are *observable*: we can print, inspect, adapt, and even transform strategies, so that teachers can use variants of a strategy, and students can customize the level of feedback when solving an exercise [12].
– Strategies are *compositional*: a strategy can be reused verbatim in another strategy.

Our strategy language satisfies these requirements to a large extent, but we have encountered a number of cases where some of the above requirements are not fulfilled. These cases are related to the cognitive fidelity principle and the compositionality requirement. For example, when solving the equation $x^2(2x^2 - 1) = 4(2x^2 - 1)$, the textbook procedure says: apply the rule $AC{=}BC \Rightarrow A{=}B \vee C{=}0$ to obtain the two quadratic equations $x^2 = 4$ and $2x^2 - 1 = 0$, and then solve these equations. So clearly the strategy for solving quadratic equations is reused in the strategy for solving equations of a higher-degree. When presenting a solution to a student, we want to first solve one quadratic equation, and then the other. But a student may solve the two equations in any order, and even switch halfway from one equation to the other. At the moment it is very hard to satisfy both requirements: our strategy for higher-degree equations does reuse the quadratic strategy, but it does not satisfy the cognitive fidelity principle. The main cause for this is the fact that we lack a language component for expressing that two strategies have to be solved, but the order in which they are solved does not matter, and steps for solving the strategies may be interleaved. For such functionality, we need an *interleaving* combinator for strategies.

Interleaving is a common operator in communicating sequential processes (CSP) [14]. It also appears under the names parallel and merge [4], but we prefer the name interleave, because it best describes the semantics of the operator we need. Parallel normally suggests that actions are performed simultaneously, which is not important in our case. In this paper we show how to extend our strategy language with several constructs to interleave strategies. The main interleave strategy combinator takes two strategies as argument, and allows a student to take steps from either of the two strategies, and finishes whenever both argument strategies are finished. We describe the semantics of the added constructs, derive properties for them, and show how they are implemented in our framework. The implementation is rather challenging because of the presence of so-called "administrative rules", which are rules that are silently applied (e.g., for navigating through a term). These rules are essential for our framework, but are not directly derived from user actions. With the extended strategy language we can more easily compose strategies, write strategies in a more natural fashion, and provide better feedback to students. The contributions of this paper are an explanation of the importance of adding facilities for specifying interleaving to a rewrite strategy language for exercises, and an implementation of the extended strategy language as a parser that recognizes student behavior and gives feedback.

This paper is organized as follows. In Section 2 we explain the need for interleaved strategies by means of several examples. Section 3 adds an interleaving strategy combinator to our strategy language, and gives its semantics. Section 4 shows how to implement an interleaving combinator for strategies in our framework, after which Section 5 discusses several design decisions about dealing with administrative rules. Section 6 shows how the added combinators help in formulating strategies for our examples. Section 7 discusses related and future work, and concludes.

2 Examples

For many exercises it is essential that we can specify that two (or more) strategies should be interleaved. This section gives examples, and discusses why existing strategy language constructs are not sufficiently expressive for these exercises.

Example: applying tautologies. In many courses on logic students are asked to rewrite logic expressions to some normal form, such as the disjunctive normal form (DNF). There are several procedures for rewriting a logic expression to DNF: one is to propagate all occurrences of the constants *true* and *false*, then replace implications and equivalences by their definitions, push negations top-down inside to the leaves of the logic expression using the rules for negation, and, finally, distribute $\wedge$ over $\vee$ to reach DNF. Furthermore, whenever a tautology or contradiction appears, simplify the formula by turning it into a constant. The first four steps are nicely captured in a strategy, but we need special machinery to model the last step for tautologies and contradictions. We could replace every rule that appears in the first four steps by the choice of that rule and the rules for tautologies and contradictions. However, such a transformation is not compositional, and bloats up strategies into huge artifacts that are hard to understand and maintain.

Example: solving polynomial equations of higher-degree. Suppose a student solves higher-degree equations in an interactive learning environment. Some higher-degree equations can be solved by applying the rule $AC{=}BC \Rightarrow A{=}B \vee C{=}0$, where the equations in the right-hand side of the rule have at most degree two. Of course, it does not matter in which order the student solves the resulting equations $A = B$ and $C = 0$, and she might even switch between solving the equations halfway. However, when we present a solution to a student, we prefer not to switch between solving the two equations. How do we describe a strategy that accepts this student behavior, and still gives hints to the student when she asks for it? We can easily express that a student should first solve the first equation and then the second, or vice versa, but this disallows switching between the two equations halfway. At the moment, we use a flexible strategy to solve these equations, which expresses that any rule from the quadratic strategy can be applied anywhere in the two equations. Since this leads to many choices, we have specified an order on the rules, and preferred rules are applied first. As a

consequence, our worked-out solution interleaves steps in solving the two equations, but sometimes in a non-intuitive way. The strategy violates the cognitive fidelity principle because not all solutions reflect the textbook description. The fact that we use a flexible strategy which accepts any rule from a particular set of rules violates the compositionality requirement: although we instruct students to apply the rule $AC{=}BC \Rightarrow A{=}B \vee C{=}0$ and then solve the resulting quadratic equations, we do not reuse the strategy for quadratic equations because that would disallow some student behavior. In this case, it was possible to relax the strategy for solving quadratic equations, such that students are allowed to switch between multiple equations that come from a single higher-degree equation. In more complicated situations, this approach might no longer be feasible.

More examples. We have used our strategy framework to develop an intelligent tutoring system for learning functional programming [10]. The tool supports the gradual refinement of programs until a program is determined to be equivalent to a model solution. Programs under development may contain one or more "holes" that need to be further refined. A student can refine these holes in any order, and a refinement step can introduce new holes. Again, to follow and support this behavior, we need a way to specify that an exercise consists of a number of strategies that can be solved in an interleaving fashion.

Other example applications which would profit from an interleaving construct for strategies are found in specifying rules for cleaning up expressions after a rewrite rule has been applied, and in tools for teaching theorem proving [16]. If such a tool has to follow the actions of a student and allow her to work on any of the subtrees to be proven, then interleaved strategies are needed for exactly the same reasons as for the functional programming tutor. Interleaving also solves the simpler problem of specifying a strategy that requires a number of rules to be applied once, but the order is irrelevant. At the moment we specify this by repeating the rules until they cannot be applied anymore, which often amounts to the same thing, but which will not work in more complicated situations.

Until now we have developed our strategies without interleaving constructs for strategies. This has led to strategies that are less precise or too strict, and harder to maintain, reuse, and adapt. Sometimes, this leads to problems in using our tools. For example, the "applying tautologies" example described above appears at the top of the list of suggested improvements to our tool for rewriting logic expressions to DNF. It is possible to specify the interleaving of strategies explicitly, but this would give huge strategies: the text size of explicitly specifying the interleaving of two strategies is more than exponential in the text size of the two argument strategies. Concluding, we need to add an interleaving strategy combinator to our strategy language.

3 Interleaving and Rewrite Strategies

In this section we show how to add interleaving to our strategy language. We will first explore the concepts of interleaving and atomicity, after which we extend

the strategy language. The notation we adopt, for interleaving and for rewrite strategies, is inspired by the algebra of communicating processes (ACP) [4]. The concise syntax makes it suitable for the type of specifications found in this paper (compared to the implementation-oriented syntax used in other papers [13, 12]). Although we use a mathematical notation in the rest of this paper, the definitions directly correspond to programs in a functional programming language like Haskell [19], and we use Haskell's semantics for recursive equations defining functions.

3.1 Interleaving Sentences

We use $a, b, c, ...$ to denote symbols, and x, y, z for sentences (sequences) of such symbols. As usual, we write ϵ for the empty sequence, and xy (or ax) for concatenation. We start by defining the interleaving of two sentences ($x \parallel y$): this operator can be defined conveniently in terms of left-interleave (denoted by $x \lfloor\!\lfloor y$, and also known as the left-merge operator [4]), which expresses that the first symbol should be taken from the left-hand side operand. ACP traditionally defines interleave in terms of left-interleave (and "communication interleave") to obtain a sound and complete axiomatization for the algebra of communicating processes [9].

$$
\begin{array}{rcl}
\epsilon \parallel x & = & \{x\} \\
x \parallel \epsilon & = & \{x\} \\
x \parallel y & = & x \lfloor\!\lfloor y \cup y \lfloor\!\lfloor x \quad (x \neq \epsilon \wedge y \neq \epsilon)
\end{array}
\qquad
\begin{array}{rcl}
\epsilon \lfloor\!\lfloor y & = & \emptyset \\
ax \lfloor\!\lfloor y & = & \{\, az \mid z \in x \parallel y \,\}
\end{array}
$$

For example, the result of interleaving the sentences abc and de (that is, $abc \parallel de$) results in the following set:

$$\{abcde, abdce, abdec, adbce, adbec, adebc, dabce, dabec, daebc, deabc\}$$

The set $abc \lfloor\!\lfloor de$ only contains the six sentences that start with symbol a. It is worth noting that the number of interleavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$. This number grows quickly with longer sentences. An alternative definition of interleaving two sequences, presented by Hoare in his influential book on CSP [14], is by means of three laws:

$$
\begin{array}{rcl}
\epsilon \in (y \parallel z) & \Leftrightarrow & y = z = \epsilon \\
x \in (y \parallel z) & \Leftrightarrow & x \in (z \parallel y) \\
ax \in (y \parallel z) & \Leftrightarrow & (\exists\, y' : y = ay' \wedge x \in (y' \parallel z)) \\
 & \vee & (\exists\, z' : z = az' \wedge x \in (y \parallel z'))
\end{array}
$$

3.2 Interleaving Sets

The operations for interleaving sentences can be lifted to work on sets of sentences by considering all combinations of elements from the two sets. Let X, Y, and Z be sets of sentences. The lifted operators are defined as follows:

$$
\begin{array}{rcl}
X \parallel Y & = & \bigcup \{\, x \parallel y \mid x \in X, y \in Y \,\} \\
X \lfloor\!\lfloor Y & = & \bigcup \{\, x \lfloor\!\lfloor y \mid x \in X, y \in Y \,\}
\end{array}
$$

For instance, $\{a, ab\} \parallel \{c, cd\}$ yields a set containing 14 elements:

$$\{abc, abcd, ac, acb, acbd, acd, acdb, ca, cab, cabd, cad, cadb, cda, cdab\}$$

From these definitions, it follows that the lifted operator for interleaving is commutative, associative, and has $\{\epsilon\}$ as identity element. The left-interleave operator is not commutative nor associative, but has the interesting property that $(X \lfloor\!\lfloor Y) \lfloor\!\lfloor Z$ is equal to $X \lfloor\!\lfloor (Y \parallel Z)$.

3.3 Atomicity

In the case of rewrite strategies, it is useful to have a notion of atomic blocks within sentences. In such a block, no interleaving should occur with other sentences. We write $\langle x \rangle$ to make sequence x atomic: if x is a singleton, the angle brackets may be dropped. Atomicity obeys some simple laws:

$$\begin{array}{rcll} \langle \epsilon \rangle & = & \epsilon & \text{(the empty sequence is atomic)} \\ \langle a \rangle & = & a & \text{(all primitive symbols are atomic)} \\ \langle x \langle y \rangle z \rangle & = & \langle xyz \rangle & \text{(nesting of atomic blocks has no effect)} \end{array}$$

In particular, it follows that $\langle\langle x \rangle\rangle = \langle x \rangle$. Atomic blocks nicely work together with the definitions given for the interleaving operators, including the lifted operators: sentences now consist of a sequence of atomic blocks, where each block itself is a non-empty sequence of symbols. For instance, $a\langle bc\rangle \parallel \langle de\rangle f$ will return:

$$\{a\langle bc\rangle\langle de\rangle f, a\langle de\rangle\langle bc\rangle f, a\langle de\rangle f\langle bc\rangle, \langle de\rangle a\langle bc\rangle f, \langle de\rangle af\langle bc\rangle, \langle de\rangle fa\langle bc\rangle\}$$

In the end, when no more interleaving takes place, the blocks have no longer any meaning, and can be discarded.

Permuting sentences (i.e., enumerating all different orderings of a list of sentences, and concatenating these sentences) can be thought of as a simpler form of interleaving. More specifically, the sentences themselves should not be interleaved, which can be done by making the sentences atomic. Hence, we define *permute* $[x_1, ..., x_n]$ as $\langle x_1 \rangle \parallel ... \parallel \langle x_n \rangle$. For example,

$$\mathit{permute}\,[ab, cde, f] = \{\,abcdef, abfcde, cdeabf, cdefab, fabcde, fcdeab\,\}$$

3.4 Interleaving Strategies

A rewrite strategy is a context free grammar with rewrite rules as terminal symbols. A rewrite strategy is defined in terms of *strategy combinators*, and is described by the following grammar:

$$\sigma \;::=\; 0 \;\mid\; 1 \;\mid\; r \;\mid\; \sigma + \sigma \;\mid\; \sigma \cdot \sigma \;\mid\; \mu\, f_\sigma \;\mid\; \ell\, \sigma$$

The basic components (symbols) of our language are rewrite rules r. Two (sub)-strategies can be combined into a strategy using the choice $(+)$ or sequence $(\cdot)$

combinator, with 0 (always fails) and 1 (always succeeds) as its unit element, respectively. The main purpose of our strategy language is to track student behavior, and to automatically calculate feedback based on the strategy and the current term. For this purpose we need to mark positions in the strategy, for which we use labels (ℓ). Such a label can, for example, be associated with a feedback text related to its particular position in the strategy.

Strategies can have recursive parts, at arbitrary positions. We use the fixpoint operator $\mu\ f_\sigma = f_\sigma\ (\mu\ f_\sigma)$ for this, where f_σ is a function that takes a strategy and returns one. With this operator, numerous derived combinators can be added to the strategy language, such as $\mathit{many}\ \sigma = \mu x.1 + \sigma \cdot x$.

The *language* (or semantics) of a strategy is a set of sentences, where each sentence is a sequence of (atomic blocks of) rewrite rules. Function $\mathcal{L}$ generates the language of a strategy, by interpreting it as a context-free grammar.

$$\begin{array}{lcllcl}
\mathcal{L}\ (0) & = & \emptyset & \mathcal{L}\ (\sigma_1 + \sigma_2) & = & \mathcal{L}\ (\sigma_1) \cup \mathcal{L}\ (\sigma_2) \\
\mathcal{L}\ (1) & = & \{\epsilon\} & \mathcal{L}\ (\sigma_1 \cdot \sigma_2) & = & \{\, xy \mid x \in \mathcal{L}\ (\sigma_1), y \in \mathcal{L}\ (\sigma_2) \,\} \\
\mathcal{L}\ (r) & = & \{r\} & \mathcal{L}\ (\mu\ f_\sigma) & = & \mathcal{L}\ (f_\sigma\ (\mu\ f_\sigma)) \\
 & & & \mathcal{L}\ (\ell\ \sigma) & = & \{\, \textsc{Enter}_{(\ell)}\ x\ \textsc{Exit}_{(\ell)} \mid x \in \mathcal{L}\ (\sigma) \,\}
\end{array}$$

With this semantics, it is easy to verify that the combinators $(+)$ and $(\cdot)$ form a semiring, as one would expect. This interpretation introduces the special rules Enter and Exit (parameterized by some label ℓ) that show up in sentences. These rules are used to trace positions in strategies. In Section 5 we discuss the subtleties of labels in strategies.

We extend the strategy language with new constructs for atomicity, interleaving, and left-interleaving:

$$\sigma\ ::=\ \ldots\ \mid\ \langle\sigma\rangle\ \mid\ \sigma \parallel \sigma\ \mid\ \sigma \between \sigma$$

The semantics for the new constructs is defined in terms of the lifted operators:

$$\begin{array}{lcl}
\mathcal{L}\ (\langle\sigma\rangle) & = & \{\, \langle x\rangle \mid x \in \mathcal{L}\ (\sigma) \,\} \\
\mathcal{L}\ (\sigma_1 \parallel \sigma_2) & = & \mathcal{L}\ (\sigma_1) \parallel \mathcal{L}\ (\sigma_2) \\
\mathcal{L}\ (\sigma_1 \between \sigma_2) & = & \mathcal{L}\ (\sigma_1) \between \mathcal{L}\ (\sigma_2)
\end{array}$$

Of course, more variations of interleaving can be added to the strategy language in a similar fashion, such as a combinator for permuting strategies. A second example is a variant of interleave that always takes steps from the left-hand side strategy if this is possible (and only if this fails, steps from the right operand), and finishes when no more steps can be done on either side.

The interleaving strategy combinator inherits the properties of the lifted interleaving operator that works on sets: it is commutative and associative, and has 1 as identity element. Because interleaving distributes over choice (that is, $\sigma_1 \parallel (\sigma_2 + \sigma_3) = (\sigma_1 \parallel \sigma_2) + (\sigma_1 \parallel \sigma_3)$), we have a second semiring. Also left-interleave distributes over choice. The operator that makes a strategy atomic is idempotent, and distributes over choice $\langle\sigma_1 + \sigma_2\rangle = \langle\sigma_1\rangle + \langle\sigma_2\rangle$. Many more properties can be found in the literature on ACP [4]. We use the properties of the strategy combinators for several purposes:

– Our implementation can be tested against these properties, and we have done so using the QuickCheck tool [6].
– The properties help strategy writers to reason about their strategies, and it provides insight into how the combinators behave.
– They will prove useful in defining the strategy recognizer, which is the topic of the next section.

4 Implementing Interleaving Strategies

Section 3 defines a language to specify rewrite strategies for exercises, extended with interleaving combinators. The definition of the semantics of this language is not suitable for implementing feedback services such as following the behavior of students, and giving hints and worked-out examples. For this, we need to develop a parser that can recognize student actions, give the next expected symbol when a student asks for a hint, or generate a complete worked-out example.

For recognizing sentences, it is sufficient to define the functions *empty* and *firsts* [13]. With these functions, input symbols can be consumed one after another, from left to right. Before we discuss how to implement the functions for the extended strategy language, we first have a look at an alternative specification for the interleaving combinator from an "operational" perspective.

We have three scenarios for parsing the strategy $\sigma_1 \parallel \sigma_2$: start with input for σ_1 (represented by $\sigma_1 \between \sigma_2$), start with σ_2, or test for the empty sentence.

$$\mathcal{L}\ (\sigma_1 \parallel \sigma_2)\ =\ \mathcal{L}\ (\sigma_1 \between \sigma_2) \cup \mathcal{L}\ (\sigma_2 \between \sigma_1) \cup \{\epsilon \mid \epsilon \in \mathcal{L}\ (\sigma_1) \cap \mathcal{L}\ (\sigma_2)\}$$

In this definition interleaving stops only when both strategies have the empty sentence, which is what the first law in Hoare's definition expresses.

4.1 Defining *empty*

The function *empty* tests whether or not the empty sentence is generated by a strategy: $empty\ (\sigma) = \epsilon \in \mathcal{L}\ (\sigma)$. The direct translation of this specification of *empty* to a functional program, using the definition of language $\mathcal{L}$, gives a very inefficient program. Instead, we derive the following recursive function from this characterization, by performing case analysis on strategies:

$$\begin{array}{lcllcl}
empty\ (0) & = & false & empty\ (\sigma_1 + \sigma_2) & = & empty\ \sigma_1 \vee empty\ \sigma_2 \\
empty\ (1) & = & true & empty\ (\sigma_1 \cdot \sigma_2) & = & empty\ \sigma_1 \wedge empty\ \sigma_2 \\
empty\ (r) & = & false & empty\ (\langle\sigma\rangle) & = & empty\ \sigma \\
empty\ (\mu\ f_\sigma) & = & empty\ (f_\sigma\ 0) & empty\ (\sigma_1 \parallel \sigma_2) & = & empty\ \sigma_1 \wedge empty\ \sigma_2 \\
empty\ (\ell\ \sigma) & = & false & empty\ (\sigma_1 \between \sigma_2) & = & false
\end{array}$$

These equations follow almost directly from the specification of $\mathcal{L}$. There is no need to visit the recursive parts to determine the *empty* property for a strategy. The definition makes explicit that the left-interleave combinator never yields the empty sentence. The new definition for $\mathcal{L}\ (\sigma_1 \parallel \sigma_2)$ shows that both σ_1 and σ_2

need to have the empty property, otherwise $\epsilon \notin \mathcal{L}\ (\sigma_1 \parallel \sigma_2)$. Interpreting these equations for *empty* as a Haskell program gives an efficient program that is linear in the size of the argument strategy.

4.2 Defining *firsts*

Given some strategy σ, the function *firsts* returns every rule that can start a sentence for σ, paired with a strategy that represents the remainder of that sentence. This is made more precise in the following specification (where r represents a rule, and x a sequence of rules):

$$\forall r, x:\ rx \in \mathcal{L}\ (\sigma) \ \Leftrightarrow\ \exists \sigma' : (r, \sigma') \in \mathit{firsts}\ (\sigma) \wedge x \in \mathcal{L}\ (\sigma')$$

As for the function *empty*, the direct translation of this specification into a functional program is infeasible. We derive an efficient implementation for *firsts* by performing a case analysis on strategies. The *firsts* set for the left-interleave case is somewhat challenging: this is exactly where we must deal with interleaving and atomicity. For a strategy $\sigma_1 \between \sigma_2$, we split σ_1 into an atomic part and a remainder, i.e., $\langle \sigma_1' \rangle \cdot \sigma_1''$. After σ_1' without the empty sentence, we can continue with $\sigma_1'' \parallel \sigma_2$. This approach is summarized by the following property, where the use of rule r takes care of the non-empty condition:

$$(\langle r \cdot \sigma_1 \rangle \cdot \sigma_2) \between \sigma_3\ =\ \langle r \cdot \sigma_1 \rangle \cdot (\sigma_2 \parallel \sigma_3)$$

Function *split* transforms a strategy into alternatives of the form $\langle r \cdot \sigma_1 \rangle \cdot \sigma_2$:

$$\begin{array}{lcl}
\mathit{split}\ (0) & = & \emptyset \\
\mathit{split}\ (1) & = & \emptyset \\
\mathit{split}\ (r) & = & \{\, \langle r \cdot 1 \rangle \cdot 1 \,\} \\
\mathit{split}\ (\sigma_1 + \sigma_2) & = & \mathit{split}\ \sigma_1 \cup \mathit{split}\ \sigma_2 \\
\mathit{split}\ (\sigma_1 \cdot \sigma_2) & = & \{\, \langle r \cdot x \rangle \cdot (y \cdot \sigma_2) \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma_1 \,\} \\
 & \cup & \textbf{if}\ \mathit{empty}\ \sigma_1\ \textbf{then}\ \mathit{split}\ \sigma_2\ \textbf{else}\ \emptyset \\
\mathit{split}\ (\mu\ f_\sigma) & = & \mathit{split}\ (f_\sigma\ (\mu\ f_\sigma)) \\
\mathit{split}\ (\ell\ \sigma) & = & \mathit{split}\ (\textsc{Enter}_{(\ell)} \cdot \sigma \cdot \textsc{Exit}_{(\ell)}) \\
\mathit{split}\ (\langle \sigma \rangle) & = & \{\, \langle r \cdot (x \cdot y) \rangle \cdot 1 \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma \,\} \\
\mathit{split}\ (\sigma_1 \parallel \sigma_2) & = & \mathit{split}\ (\sigma_1 \between \sigma_2) \cup \mathit{split}\ (\sigma_2 \between \sigma_1) \\
\mathit{split}\ (\sigma_1 \between \sigma_2) & = & \{\, \langle r \cdot x \rangle \cdot (y \parallel \sigma_2) \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma_1 \,\}
\end{array}$$

We briefly discuss the definitions for the new constructs:

- *Case* $\langle \sigma \rangle$: because atomicity distributes over choice, we can consider the elements of *split* σ (the recursive call) one by one. The transformation $\langle \langle r \cdot x \rangle \cdot y \rangle = \langle r \cdot (x \cdot y) \rangle \cdot 1$ is proven by first removing the inner atomic block, and basic properties of sequence.
- *Case* $\sigma_1 \parallel \sigma_2$: expressing this strategy in terms of left-interleave is justified by the definition of $\mathcal{L}\ (\sigma_1 \parallel \sigma_2)$ given in this section. For function *split*, we only have to consider the non-empty sentences.

– *Case* $\sigma_1 \mathbin{\lfloor\!\lfloor} \sigma_2$: left-interleave can be distributed over the alternatives. Furthermore, $(\langle r \cdot x \rangle \cdot y) \mathbin{\lfloor\!\lfloor} \sigma_2 = \langle r \cdot x \rangle \cdot (y \parallel \sigma_2)$ follows from the definition of left-interleave on sentences (with atomic blocks).

With the function *split*, we can now define the function *firsts* needed for most of our feedback services:

$$\mathit{firsts}\ (\sigma) \ = \ \{\, (r, x \cdot y) \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma \,\}$$

5 Dealing with Administrative Rules

Our strategy framework uses *administrative rules* to change the context of an expression, but not the expression itself. Students cannot observe the application of administrative rules. Examples of such rules are tracking the labels in a strategy, keeping a focus (on a subexpression), and reading (or writing) a value from an environment. This section describes practical issues with administrative rules that arise when adding interleaving to the strategy language. We discuss how to deal with labels and navigation actions for moving the point in focus.

5.1 Labels in Strategies

Consider the sentences generated by the following strategy:

$$\ell_1\ (r_1 \cdot \ell_2\ (r_2 \cdot r_3) \parallel \ell_3\ (r_4 \cdot r_5))$$

When ignoring labels, this strategy generates 10 ($= \frac{5!}{2!3!}$) sentences. For each labeled (sub)strategy, we insert administrative rules to mark where we enter or leave that strategy: $\ell\ \sigma$ thus becomes $\textsc{Enter}_{(\ell)} \cdot \sigma \cdot \textsc{Exit}_{(\ell)}$. These markings tell the framework where a student is in a strategy, and allows us to give appropriate feedback based on the labels. However, the extra steps also significantly increase the number of sentences ($\frac{11!}{4!7!} = 330$). This explosion in the number of sentences quickly makes the strategy unusable for the purpose of generating feedback and tracking student behavior: there are too many possibilities to choose from.

Since users cannot observe administrative rules, the sentences with the administrative rules do not add interleavings that are interesting for a user. It does not make sense to switch to another interleaved strategy right after an enter or exit step, and before a major (non-administrative) step is detected. Hence, we allow the prefixes $\textsc{Enter}_{(\ell_1)} r_1$ and $\textsc{Enter}_{(\ell_3)} r_4$, but we disallow $\textsc{Enter}_{(\ell_1)}\textsc{Enter}_{(\ell_3)}$ and $\textsc{Enter}_{(\ell_3)}\textsc{Enter}_{(\ell_1)}$. This gives us again 10 sentences.

As a consequence of the administrative rules, we need variants of *empty* and *firsts* that skip over these rules and behave properly in the presence of interleaved parts, along the lines of the big step operator defined by Gerdes et al. [11]. For administrative rule r we have that $r \cdot \langle \sigma \rangle$ can be transformed into $\langle r \cdot \sigma \rangle$. This property can be used to refine the *split* function for the left-interleave case.

5.2 Navigation Actions

Many rewrite strategies in mathematics rely on navigation combinators that move the focus to a particular subexpression. For this, we use the zipper data-structure [15], and its operations such as moving up and down in a tree. In our strategies, we use the administrative rules Up and Downs. The rule for moving downwards returns multiple (zero or more) alternatives with a new focus, one for each child. With these navigation rules, we can define the *somewhere* combinator:

$$somewhere\ \sigma = \mu x.\sigma + (\textsc{Downs} \cdot x \cdot \textsc{Up})$$

Again, interleaving poses an additional challenge when dealing with administrative rules, this time for navigation. Consider the strategy $(somewhere\ \sigma_1) \parallel \sigma_2$, and assume that the rules in σ_2 take the current focus into account. It is undesirable to interleave the navigation actions from the left-hand side strategy with σ_2. This gives highly unpredictable behavior, especially when σ_2 also performs navigation actions. Therefore, we make the result of *somewhere* atomic by default. When moving the focus down, all interleaved strategies are blocked until the matching up action is recognized. We do the same for the other navigation combinators (e.g., *topDown*, which applies a strategy at the highest possible position in a tree). Note that a *somewhere* combinator that does allow interleaving (for instance, because the other strategy is known to ignore the focus) can still be defined if desired.

Besides navigation, there are other ways in which rules of interleaved strategies can interfere. Examples are rules that read and write values to an environment, or a rule that assumes a certain invariant to hold when it is executed. For such cases, the strategy developer has to make parts of the strategy atomic, or take other measures to ensure non-interference. This sometimes makes developing strategies significantly more complex, which is not uncommon when concurrency is involved.

6 Examples Revisited

This section revisits the examples given in Section 2, and shows how we can define strategies for these examples using the interleaving combinators.

6.1 Applying Tautologies

Figure 1 presents a collection of rules for rewriting logical expressions. We assume that these rules are applied from left to right. Rules for expressing the associativity of conjunction and disjunction are missing: instead, we assume that the given rules are applied modulo the associativity of these operators. The rule set should also be completed by adding commutative variations of the presented rewrite rules (e.g., $F \vee \phi = \phi$ for OrFalse).

The rules in Figure 1 are grouped into categories (such as "negations"), and we use this grouping to combine the rules and categories into strategies:

Basic Rules:

Constants:	AndTrue:	$\phi \wedge T = \phi$	AndFalse:	$\phi \wedge F = F$
	OrTrue:	$\phi \vee T = T$	OrFalse:	$\phi \vee F = \phi$
	NotTrue:	$\neg T = F$	NotFalse:	$\neg F = T$
Definitions:	ImplDef:	$\phi \rightarrow \psi = \neg\phi \vee \psi$		
	EquivDef:	$\phi \leftrightarrow \psi = (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$		
Negations:	NotNot:	$\neg\neg\phi = \phi$		
	DeMorganAnd:	$\neg(\phi \wedge \psi) = \neg\phi \vee \neg\psi$		
	DeMorganOr:	$\neg(\phi \vee \psi) = \neg\phi \wedge \neg\psi$		
Distribution:	AndOverOr:	$\phi \wedge (\psi \vee \chi) = (\phi \wedge \psi) \vee (\phi \wedge \chi)$		

Additional Rules:

Tautologies:	ImplTaut:	$\phi \rightarrow \phi = T$	OrTaut:	$\phi \vee \neg\phi = T$
	EquivTaut:	$\phi \leftrightarrow \phi = T$		
Contradictions:	AndContr:	$\phi \wedge \neg\phi = F$	EquivContr:	$\phi \leftrightarrow \neg\phi = F$

Fig. 1. Rules for logical expressions

$$
\begin{aligned}
\mathit{negations} &= \textsc{NotNot} + \textsc{DeMorganAnd} + \textsc{DeMorganOr} \\
\mathit{basics} &= \mathit{constants} + \mathit{definitions} + \mathit{negations} + \mathit{distribution} \\
\mathit{additionals} &= \mathit{tautologies} + \mathit{contradictions}
\end{aligned}
$$

A straightforward approach to reach disjunctive normal form (DNF) is to apply the basic rules exhaustively (the *repeat* combinator), where *somewhere* makes sure that the rules can also be applied to subexpressions:

$$\mathit{dnfExhaustive} = \mathit{repeat}\ (\mathit{somewhere}\ \mathit{basics})$$

This strategy is very liberal, and also generates derivations that are less intuitive. The following refined strategy proceeds in four steps, and makes more precise which rule should be applied when, and where.

$$
\begin{aligned}
\mathit{dnfSteps} = {} & \mathit{label}\ \texttt{"constants"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants}\ )) \\
& \cdot\ \mathit{label}\ \texttt{"definitions"}\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{definitions}\ )) \\
& \cdot\ \mathit{label}\ \texttt{"negations"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{negations}\ )) \\
& \cdot\ \mathit{label}\ \texttt{"distribution"}\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{distribution}))
\end{aligned}
$$

For example, consider applying strategy *dnfSteps* to $\neg((p \vee q) \rightarrow p)$. This results in the following derivation (and for this logical proposition, no other derivation):

$$
\begin{aligned}
\neg((p \vee q) \rightarrow p) &= \neg(\neg(p \vee q) \vee p) && (\textsc{ImplDef}) \\
&= \neg\neg(p \vee q) \wedge \neg p && (\textsc{DeMorganOr}) \\
&= (p \vee q) \wedge \neg p && (\textsc{NotNot}) \\
&= (p \wedge \neg p) \vee (q \wedge \neg p) && (\textsc{AndOverOr})
\end{aligned}
$$

If we would have used strategy *dnfExhaustive*, many more derivations would have been allowed, including derivations where $\neg\neg(p \vee q)$ is rewritten into $\neg(\neg p \wedge \neg q)$, giving seven steps in total (instead of just four).

Suppose that we want to extend our strategies for reaching DNF, and also want to use rules for tautologies and contradictions. In the case of *dnfExhaustive*, this can be accomplished by changing its definition into:

$$dnfExtra = repeat\ (somewhere\ (basics + additionals))$$

Changing the original strategy is not always possible, for instance if you want to have the original strategy (without the extra rules) but also the extended strategy (with the extra rules). An alternative approach is to reuse *dnfExhaustive* verbatim, and define *dnfExtra* as follows:

$$dnfExtra = repeat\ (dnfExhaustive + somewhere\ additionals)$$

This definition has some disadvantages too. First of all, the strategy differs considerably from the informal description, and violates the cognitive fidelity principle. Such a difference also influences feedback. Secondly, during the execution of strategy *dnfExhaustive*, this strategy disallows applications of the additional rules. For instance, consider the proposition $p \vee \neg(p \wedge \neg q)$. After one step (taken from *dnfExhaustive*) this is rewritten into $p \vee \neg p \vee \neg\neg q$. At this point, *dnfExhaustive* is not yet finished (because of the double negation), and as a result, the strategy disallows the step with rule OrTaut to $T \vee \neg\neg q$. Extending strategy *dnfExhaustive* using the interleaving combinator is straightforward:

$$dnfExtra = label\ \texttt{"extra"}\ (repeat\ (somewhere\ additionals)) \parallel dnfExhaustive$$

The label in the strategy specification is not necessary, but it provides extra information that can be used for the generation of hints. We return to our earlier example, for which the extended strategy generates two more steps:

$$\begin{array}{lcll} \neg((p \vee q) \rightarrow p) & = & \ldots & \\ & = & (p \wedge \neg p) \vee (q \wedge \neg p) & (\textsc{AndOverOr}) \\ & = & F \vee (q \wedge \neg p) & (\textsc{AndContr}) \\ & = & q \wedge \neg p & (\textsc{OrFalse}) \end{array}$$

Observe the interleaving of steps in this derivation: AndOverOr and OrFalse originate from the *dnfExhaustive* strategy, whereas AndContr comes from the part labeled "extra". Note that this strategy also permits other derivations.

Interestingly, our last definition of *dnfExtra* with interleaving is equivalent to the simpler strategy *repeat* (*somewhere* (*basics* + *additionals*)), and this equivalence can be shown using basic properties of our strategy combinators. To be precise, we need a property explaining how two interleaved repetitions behave:[1]

[1] Since we have not defined *repeat*, we do not prove this property here. The property also holds for *many* (see Section 3.4).

repeat $\sigma_1 \parallel$ *repeat* $\sigma_2 =$ *repeat* $(\sigma_1 + \sigma_2)$. Likewise, we use that *somewhere* distributes over choice: *somewhere* $(\sigma_1 + \sigma_2) =$ *somewhere* $\sigma_1 +$ *somewhere* σ_2. This emphasizes once more the need for having a clear semantics for the combinators.

Strategy *dnfSteps* can also be extended with rules for tautologies and contradictions. Constants are removed in the first step, but the constants introduced by the new rules have to be propagated as well. Hence, the extension to the *dnfSteps* strategy not only adds the new rules, but it also takes care of dealing with the newly introduced constants. The following code fragment shows a possible definition:

$$\begin{aligned} \mathit{extension} &= \mathit{repeat}\ (\mathit{somewhere}\ (\mathit{additionals} + \mathit{constants})) \\ \mathit{dnfExtension} &= \mathit{label}\ \texttt{"extension"}\ \mathit{extension} \parallel \mathit{dnfSteps} \end{aligned}$$

Note that this definition is ambiguous in how the constants are removed, because *dnfSteps* can do this (in the first step), but also the extension. This ambiguity has no consequences for the feedback services we offer. The definition gives no priority to the extra rules: they may be used (if possible), but this is not mandatory. Also, it is not required to remove the constants that are introduced by a tautology or contradiction before continuing with bringing the logic expression to DNF.

Our strategy language is expressive enough to specify that constants have to be propagated immediately, including the ones from tautologies and contradictions. For this, we use the atomic combinator in the extension:

$$\mathit{extension} = \mathit{repeat}\ \langle \mathit{somewhere}\ \mathit{additionals} \cdot \mathit{repeat}\ (\mathit{somewhere}\ \mathit{constants}) \rangle$$

6.2 Solving Polynomial Equations of a Higher-degree

Now that we have interleaving of strategies available, we can adapt the strategy for solving higher-degree equations to make it possible to:

- reuse the strategy for solving quadratic equations;
- follow student behavior even when a student switches from solving one equation to the other;
- give hints about the equation the student is currently solving; and
- show worked-out solutions in which first one of the two quadratic equations is solved, and then the other.

For the latter two points, strategies need to be labeled. The labels are used to determine where in a strategy a student is, and what the corresponding first step would be. Labeling a strategy can either be done automatically, or we can leave this to the strategy developer.

7 Conclusions and Related Work

We have shown how we can add interleaving combinators to our language for specifying rewrite strategies for exercises. We have implemented these combinators in our framework, such that we can follow student behavior, give hints, and

show worked-out examples. The implementation of the new combinators, discussed in Section 4, can be translated almost literally to executable Haskell code, the programming language of our choice. The upcoming release of our framework on Hackage[2] will contain the new combinators. Using the interleaving combinators, we can specify strategies closer to textbook description of strategies, allow for more natural student behavior, specify more strategies compositionally, and give various kinds of feedback for strategies using the interleaving combinators. This makes it easier to develop, use, maintain, and reuse strategies.

Our strategy language [13] is similar to strategy languages used in computer science and theorem proving [21, 5, 1] extended with constructs that support giving feedback, such as labels (also present in [1]) and navigation. An interleaving combinator for tactics is easily implemented in a theorem prover such as Isabelle [18]. The interleaving combinators are inspired by the work on communicating sequential processes (CSP) and the algebra of communicating processes (ACP) [14, 4], but our goal is to model interactive exercises and to give feedback, instead of modeling concurrent processes. The differences between the ACP approach and our work is the interpretation of a strategy as a parser, which can deal with administrative rules, and the introduction of an operator for specifying atomicity. In contrast with ACP and CSP, we have found no need for adding a communication operator to our language. Our implementation can be seen as an "interleaving parser", which adds an extra level of (interleaving) complexity on top of "permutation parsers" [2], which can be used to parse a number of elements in any order. Current parsing combinator libraries do not offer parser combinators for interleaving parsers.[3]

We have implemented interleaving in our framework, but we have yet to gain large-scale experience with the combinators. We will include the interleaving combinator in several of our strategies used within the Math-Bridge project[4], and evaluate the results.

Acknowledgments. This work was made possible by the Math-Bridge project of the Community programme eContent*plus*. The paper does not represent the opinion of the Community, and the Community is not responsible for any use that might be made of information contained in this paper. We thank Alex Gerdes and the anonymous reviewers for commenting on an earlier version of this paper.

References

1. D. Aspinall, E. Denney, and C. Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3(3):309–330, 2010.

[2] `http://hackage.haskell.org/package/ideas`

[3] After discussing our work with Doaitse Swierstra, he implemented a (rather involved) interleaving combinator for parsers on top of his parser combinator library [20].

[4] `http://service.math-bridge.org/`

2. A.I. Baars, A. Löh, and S.D. Swierstra. Parsing permutation phrases. *Journal of Functional Programming*, 14:635–646, November 2004.
3. M.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, 1998.
4. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77 – 121, 1985.
5. A. Bundy. The use of explicit plans to guide inductive proofs. In *International conference on automated deduction*, pages 111–120, 1988.
6. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00*, pages 268–279, 2000.
7. A. Cohen, H. Cuypers, E. Reinaldo Barreiro, and H. Sterk. Interactive mathematical documents on the web. In *Algebra, Geometry and Software Systems*, pages 289–306. Springer, 2003.
8. M. Doorman, P. Drijvers, P. Boon, S. van Gisbergen, and K. Gravemeijer. Design and implementation of a computer supported learning environment for mathematics. In *Earli 2009 SIG20 invited Symposium Issues in designing and implementing computer supported inquiry learning environments*, 2009.
9. W. Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 2000.
10. A. Gerdes, B. Heeren, and J. Jeuring. Constructing Strategies for Programming. In J. Cordeiro et al., editor, *Proceedings of the First International Conference on Computer Supported Education*, pages 65–72. INSTICC Press, March 2009.
11. A. Gerdes, B. Heeren, and J. Jeuring. Properties of Exercise Strategies. In *Proceedings of IWS 2010: 1st International Workshop on Strategies in Rewriting, Proving, and Programming, Electronic Proceedings in Theoretical Computer Science*, 2011.
12. B. Heeren and J. Jeuring. Adapting mathematical domain reasoners. In *Proceedings of the 9th MKM international conference*, MKM'10, pages 315–330. Springer, 2010.
13. B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010.
14. C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.
15. G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
16. J. Lodder and B. Heeren. A teaching tool for proving equivalences between logical formulae. In *Third International Congress on Tools for Teaching Logic*, volume 6680 of *LNCS*. Springer, 2011.
17. E. Melis and J. Siekmann. ActiveMath: An intelligent tutoring system for mathematics. In *ICAISC*, volume 3070 of *LNCS*, pages 91–101. Springer, 2004.
18. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
19. S. 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/`.
20. S.D. Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300, Berlin, Heidelberg, 2009. Springer-Verlag.
21. E. Visser, Z.A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP'98*, pages 13–26, 1998.