Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 237 (2009) 91–106
www.elsevier.com/locate/entcs

Recognizing Strategies

Bastiaan Heeren[1,2]

School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands

Johan Jeuring[3]

School of Computer Science, Open Universiteit Nederland and ICS, Universiteit Utrecht

Abstract

We use strategies to specify how a wide range of exercises can be solved incrementally, such as bringing a logic proposition to disjunctive normal form, reducing a matrix, or calculating with fractions. With such a strategy, we can automatically generate worked-out solutions, track the progress of a student by inspecting submitted intermediate answers, and report back suggestions in case the student deviates from the strategy. Because we can calculate all kinds of feedback automatically from a strategy specification, it becomes less labor-intensive and less ad-hoc to specify new exercise domains and exercises within that domain.
A strategy describes valid sequences of transformation rules that solve the exercise at hand, which turns tracking intermediate steps into a parsing problem. This is a promising view at the problem because it allows us to take advantage of many years of experience in parsing sentences of context-free languages, and transfer this knowledge and technology to the domain of stepwise solving exercises.
In this paper we work out the similarities between parsing and solving exercises incrementally, and we discuss the implementation of a recognizer for strategies. We present a full implementation of such a recognizer, and discuss a number of design choices we have made. In particular, we discuss the use of a fixed point combinator to deal with repetition, and labels to mark positions in the strategy.

Keywords: grammars, parsing, strategies, exercise assistants, combinator languages

1 Introduction

Strategies are used in many domains such as programming, rewriting, compiler construction, and theorem proving. We recently realized that strategies also play an important role in exercise assistants that support incrementally solving exercises in mathematics, logics, physics, etc. [6]. In the intelligent tutoring systems field, a strategy is called procedural knowledge, a production system, or a procedural plan.

[1] This work was made possible by the support of the SURF Foundation, the higher education and research partnership organisation for Information and Communications Technology (ICT). For more information about SURF, please visit http://www.surf.nl. We thank Alex Gerdes and the anonymous reviewers for their constructive comments.

[2] Email: bastiaan.heeren@ou.nl

[3] Email: johanj@cs.uu.nl

doi:10.1016/j.entcs.2009.03.037

In this field, strategies are not used to rewrite terms, but they are used to check that a user performs the correct steps towards a solution for an exercise.

An exercise such as "rewrite the following arithmetic expression containing fractions to its normal form", consists of the expression to be rewritten, the rules with which the expression can be rewritten (and possibly also some known buggy rules), and a strategy to guide or direct the rewriting. If a user solves the exercise incrementally, we can check at each step whether or not the step is a valid rewrite step, and whether or not this rewrite step is valid according to the strategy. In a way we check whether or not the sequence of rewrite steps performed by the user is a prefix of a sentence in the language specified by the strategy.

This paper briefly explains a language for specifying strategies for exercises, and it shows how we use this language for recognizing valid sequences of rewrite steps. The paper has two contributions:

- It discusses the design choices when constructing a strategy language for specifying exercises, and a strategy recognizer for such a language.
- It shows how we can use the strategy language to check whether or not user-input is correct with respect to the strategy specified for an exercise.

The information provided by our strategy recognizer is necessary for determining what kind of feedback to give to a user of an exercise assistant. At the moment the feedback provided by exercise assistants is almost always limited to correct/incorrect. Using the diagnosis given by our strategy recognizer we can improve a lot on this.

This paper is organized as follows. Section 2 introduces our language for specifying strategies for exercises. We illustrate the language with a strategy for adding fractions: this strategy is used as a running example throughout the paper. Section 3 shows how we have implemented the components of our strategy language to obtain a strategy recognizer, and discusses the main design choices. We then present three extensions to our strategy recognizer in Section 4. Section 5 shows how the strategy language can be used for diagnosing possible problems in the user input. The last two sections (6 and 7) discuss related work and ongoing research, and draw conclusions.

2 A strategy language

Before we introduce our strategy language, which is inspired by context-free grammars (CFG), we give an example strategy.

Example 2.1 Consider the problem of adding two fractions, for example, $\frac{2}{5}$ and $\frac{2}{3}$: if the result is an improper fraction (the numerator is larger than or equal to the denominator), then it should be converted to a mixed number. Figure 1 displays four rewrite rules on fractions. The three rules on the right (B1 to B3) are buggy rules that capture common mistakes. A possible strategy to solve this type of exercise is the following:

$$\frac{a}{c}+\frac{b}{c} = \frac{a+b}{c} \quad [\textsc{Add}]$$
$$\frac{a}{b}\times\frac{c}{d} = \frac{a\times c}{b\times d} \quad [\textsc{Mul}]$$
$$\frac{b}{c} = \frac{a\times b}{a\times c} \quad [\textsc{Rename}]$$
$$\frac{a+b}{b} = 1+\frac{a}{b} \quad [\textsc{Simpl}]$$

$$\frac{a}{b}+\frac{c}{d} \neq \frac{a+c}{b+d} \quad [\text{B1}]$$
$$a\times\frac{b}{c} \neq \frac{a\times b}{a\times c} \quad [\text{B2}]$$
$$a+\frac{b}{c} \neq \frac{a+b}{c} \quad [\text{B3}]$$

Fig. 1. Transformation rules in the domain of fractions

- *Step 1.* Find the least common denominator (lcd) of the fractions: let this be n
- *Step 2.* Rename the fractions such that n is the denominator
- *Step 3.* Add the fractions by adding the numerators
- *Step 4.* Simplify the fraction if it is improper

A context-free grammar distinguishes terminal and non-terminal symbols, and has a set of production rules. For our strategy language, we take a different approach and use combinators instead. Strategies are constructed in the following way:

- **Transformation rule.** Such a rule is the smallest building block to construct composite strategies, and closely corresponds to a terminal symbol in a CFG. Occasionally, we write *symbol* r for some transformation rule r to distinguish the strategy from the transformation rule.
- **Sequence.** We can combine strategies s and t and put them in sequence, for which we write s <*> t. A production rule in a CFG is a sequence of symbols.
- **Choice.** Another way to combine strategies is by choice: we write s <|> t for choosing between strategy s and strategy t. In CFGs, choice is introduced by having multiple production rules for a non-terminal symbol.
- **Unit elements.** We introduce two special elements that are the units for sequence and choice. The strategy *succeed* always succeeds (unit element for <*>), whereas *fail* always fails (unit element for <|>).
- **Labels.** Because our primary interest in the strategy language is to automatically calculate feedback from it, we need some mechanism to mark positions in the strategy, for example, to encode the hierarchical structure of a strategy, or to refine the textual feedback that is associated with a certain position in the strategy. For this purpose, we introduce labels. Labeling a strategy s with some label ℓ is written as *label* ℓ s. The exact representation of a label is irrelevant.
- **Recursion.** We need a way to deal with recursion, and for this we introduce a fixed point combinator. We write *fix* f, where f is the function of which we take the fixed point. Hence, the function f takes a strategy and returns one, and such that the property *fix* $f = f$ (*fix* f) holds.

Example 2.2 Repetition, zero or more occurrences of something, is a well-known recursion pattern. We can define this pattern using our fixed point recursion combinator:

$$many\ s = fix\ (\lambda x \rightarrow succeed <|> (s <\star> x))$$

The strategy that applies transformation rule r zero or more times would thus be:

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

Example 2.3 We use the strategy combinators to turn the informal strategy description from Example 2.1 into a strategy specification:

$$\begin{aligned}
addFractions = label\ \ell_0 \quad (&\quad label\ \ell_1\ ruleLCD\\
<\star>\ &label\ \ell_2\ (repeat\ (somewhere\ ruleRename))\\
<\star>\ &label\ \ell_3\ ruleAdd\\
<\star>\ &label\ \ell_4\ (try\ ruleSimpl)\\
&)
\end{aligned}$$

The strategy contains the labels ℓ_0 to ℓ_4, and uses the transformation rules given in Figure 1. The transformation *ruleLCD* is somewhat different: it does not change the term, but it calculates the least common denominator and stores this in an environment. The rule RENAME for renaming a fraction uses the computed lcd to determine the value of a in its right-hand side. Rules that do not change the term but only the context in which an exercise is solved are so-called *administrative rules*.

The definition of *addFractions* contains the strategy combinators *repeat*, *somewhere*, and *try*. In an earlier paper [6], we discussed how these combinators, and many others, can be defined conveniently in terms of the strategy language. The combinator *repeat* is a variant of the *many* combinator: it applies its argument strategy exhaustively. The check that the strategy can no longer be applied is an administrative rule. The definition of *somewhere* is another example of an administrative rule: this combinator changes the focus in the abstract syntax tree before it applies its argument strategy. The zipper data structure [8] can be used to keep a point of focus.

2.1 Semantics of the strategy language

Before we move on to the implementation, we make our strategy combinators more precise by defining the language that is generated by a strategy. Such a language is a set of sequences of transformation rules.

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

$$\begin{aligned} \mathit{language\ succeed} &= \{\epsilon\} \\ \mathit{language\ fail} &= \emptyset \end{aligned}$$

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

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

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

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

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

3 Design of a strategy recognizer

In this section we discuss the design of a strategy recognizer. Instead of designing our own recognizer, we could reuse existing parsing libraries and tools. There are many excellent parser generators and various parser combinator libraries around [9,13], and these are often highly optimized and efficient in both their time and space behavior. However, the problem we are facing is quite different from other parsing applications. To start with, efficiency is no longer a key concern. Because we are recognizing applications of rewrite rules applied by a student, the length of the input is very limited. Our experience until now is that speed poses no serious constraints on the design of the library. A second difference is that we are not building an abstract syntax tree.

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

(i) We are only interested in sequences of transformation rules that can be applied successively to some initial term, and this is hard to express in most libraries.

Parsing approaches that start by analyzing the grammar for constructing a parsing table will not work in our setting because they can not take the current term into account.

(ii) The ability to diagnose errors in the input highly influences the quality of the feedback services. It is not enough to detect that the input is incorrect, but we also want to know at which point the input deviates from the strategy, and what is expected at this point. Some of the more advanced parser tools have error correcting facilities, which helps diagnosing an error to some extent.

(iii) Exercises are solved incrementally, and therefore we do not only have to recognize full sentences, but also prefixes. Backtracking and look-ahead can not be used because we want to recognize strategies at each intermediate step.

(iv) Labels help to describe the structure of a strategy in the same way as non-terminals do in a grammar. For a good diagnosis it is vital that a recognizer knows at each intermediate step where it is in the strategy.

(v) A strategy should be serializable, for instance because we want to communicate with other e-learning tools and environments.

In earlier attempts to design a recognizer library for strategies, we tried to reuse an existing error-correcting parser combinator library [13], but failed because of the reasons listed above. The library we develop in this paper is written in the functional programming language Haskell [12]. The code in this paper is almost complete and conforms to the Haskell 98 standard. Although the code is relatively short, we want to emphasize that the library has been tested in practice on different domains. For instance, strategies implemented for the domain of linear algebra are more complex than the strategy for fractions reported in this paper. These strategies will be used in several courses during 2008.

3.1 Representing grammars

Because strategies are grammars, we start by exploring a suitable representation for grammars. The data type for grammars is based on the alternatives of the strategy language discussed in Section 2:

```
data Grammar a = Grammar a :⋆: Grammar a
               | Grammar a :|: Grammar a
               | Rec Int (Grammar a)
               | Symbol a | Var Int | Succeed | Fail deriving Show
```

The type variable a in this definition is an abstraction for the type of the symbols: for strategies, the symbols are rules. The first design choice is how to represent recursive grammars, for which we use the constructors *Rec* and *Var*. A *Rec* binds all the *Var*s in its scope that have the same integer. We assume that all our grammars are closed, i.e., there are no free occurrences of variables. This data type makes it easy to manipulate and analyze grammars. Alternative representations for recursion are higher-order fixed point functions, or nameless terms using de Bruijn

indices.

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

3.2 Smart constructors

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

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

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

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

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

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

$$
\begin{array}{l}
\mathit{rec} :: \mathit{Int} \to \mathit{Grammar}\ a \to \mathit{Grammar}\ a \\
\mathit{rec}\ i\ s = \mathbf{if}\ i \in \mathit{freeVars}\ s\ \mathbf{then}\ \mathit{Rec}\ i\ s\ \mathbf{else}\ s
\end{array}
$$

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

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

$$
\mathit{fix} :: (\mathit{Grammar}\ a \to \mathit{Grammar}\ a) \to \mathit{Grammar}\ a
$$

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

3.3 Empty and firsts

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

$$
\begin{aligned}
&\mathit{empty} :: \mathit{Grammar}\ a \to \mathit{Bool}\\
&\mathit{empty}\ (s :\star: t) &&= \mathit{empty}\ s \wedge \mathit{empty}\ t\\
&\mathit{empty}\ (s :|: t) &&= \mathit{empty}\ s \vee \mathit{empty}\ t\\
&\mathit{empty}\ (\mathit{Rec}\ i\ s) &&= \mathit{empty}\ s\\
&\mathit{empty}\ \mathit{Succeed} &&= \mathit{True}\\
&\mathit{empty}\ \_ &&= \mathit{False}
\end{aligned}
$$

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

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

$$
\begin{aligned}
&\mathit{firsts} :: \mathit{Grammar}\ a \to [(a, \mathit{Grammar}\ a)]\\
&\mathit{firsts}\ (s :\star: t) &&= [(a, s' <\star> t) \mid (a, s') \leftarrow \mathit{firsts}\ s] +\!\!+\\
&&&\quad (\textbf{if}\ \mathit{empty}\ s\ \textbf{then}\ \mathit{firsts}\ t\ \textbf{else}\ [\,])\\
&\mathit{firsts}\ (s :|: t) &&= \mathit{firsts}\ s +\!\!+ \mathit{firsts}\ t\\
&\mathit{firsts}\ (\mathit{Rec}\ i\ s) &&= \mathit{firsts}\ (\mathit{replaceVar}\ i\ (\mathit{Rec}\ i\ s)\ s)\\
&\mathit{firsts}\ (\mathit{Symbol}\ a) &&= [(a, \mathit{succeed})]\\
&\mathit{firsts}\ \_ &&= [\,]
\end{aligned}
$$

For a sequence $(s :\star: t)$, we determine which symbols can appear first for s, and we change the results to reflect that t is part of the remaining grammar. Furthermore, if s can be empty, then we also have to look at the *firsts* for t. For choices, we simply combine the results for both operands. If the grammar is a single symbol, then this symbol appears first, and the remaining strategy is *succeed* (we are done). To find the firsts for $(\mathit{Rec}\ i\ s)$, we have to look inside the body s. All occurrences of this recursion point are replaced by the grammar itself before we call *firsts* again. The replacement is performed by a helper-function: *replaceVar* $i\ s\ t$ replaces all free occurrences of $(\mathit{Var}\ i)$ in t by s.

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

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

Example 3.1 The repetition combinator *many* can be defined in the following way:

$$
\begin{aligned}
&many :: Grammar\ a \to Grammar\ a \\
&many\ s = rec\ 0\ (succeed <|> (nonempty\ s <\star> var\ 0))
\end{aligned}
$$

It can also be expressed using the function *fix*, resulting in the definition given in Example 2.2. We have to apply *nonempty* to strategy s to avoid a left-recursive grammar specification: this also holds when we use *fix*. In Section 4.2 we explain how left recursion can be avoided by analyzing the grammar that is constructed.

3.4 Running a strategy

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

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

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

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

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

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

3.5 Labels

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

$$\textbf{data}\ \mathit{LabeledStrategy}\ l\ a = \mathit{Label}\ l\ (\mathit{Strategy}\ l\ a)$$
$$\textbf{type}\ \mathit{Strategy}\quad\ \ l\ a = \mathit{Grammar}\ (\mathit{Either}\ (\mathit{Rule}\ a)\ (\mathit{LabeledStrategy}\ l\ a))$$

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

Excluding labels from the *Grammar* data type is a design choice. Functions that work on the *Grammar* data type don't have to deal with labels, which makes it, for example, simpler to manipulate grammars. A disadvantage of our solution is that symbols in a strategy must be tagged *Left* or *Right*. In our actual implementation we circumvent the tagging by overloading the strategy combinators. As a result, strategies can really be defined as the specification in Example 2.3.

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

$$\textbf{data}\ \mathit{Step}\ l\ a = \mathit{Enter}\ l \mid \mathit{Step}\ (\mathit{Rule}\ a) \mid \mathit{Exit}\ l\ \textbf{deriving}\ \mathit{Show}$$

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

$$\begin{aligned}
&\mathit{withSteps} :: \mathit{LabeledStrategy}\ l\ a \to \mathit{Grammar}\ (\mathit{Step}\ l\ a)\\
&\mathit{withSteps}\ (\mathit{Label}\ l\ s) \ = \ \mathit{symbol}\ (\mathit{Enter}\ l)\\
&\qquad\qquad\qquad\qquad <\star> \mathit{mapSymbol}\ (\mathit{either}\ (\mathit{symbol} \circ \mathit{Step})\ \mathit{withSteps})\ s\\
&\qquad\qquad\qquad\qquad <\star> \mathit{symbol}\ (\mathit{Exit}\ l)
\end{aligned}$$

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

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

Example 3.2 Suppose that we run the strategy of Example 2.3 on the term $\frac{2}{5} + \frac{2}{3}$: what would be the result? Of course, we would expect to get the derivation:

$$\frac{2}{5} + \frac{2}{3} \ = \ \frac{6}{15} + \frac{2}{3} \ = \ \frac{6}{15} + \frac{10}{15} \ = \ \frac{16}{15} \ = \ 1\frac{1}{15}$$

The final answer, $1\frac{1}{15}$, is indeed what we get. In fact, this term is returned twice because the strategy does not specify which of the fractions should be renamed first, which results in two different derivations. It is much more informative to step through the above derivation and see the intermediate steps.

$$
\begin{array}{[lllll}
\mathit{Enter}\ \ell_0, & \mathit{Enter}\ \ell_1, & \mathit{Step}\ \mathit{ruleLCD}, & \mathit{Exit}\ \ell_1, & \mathit{Enter}\ \ell_2, \\
\mathit{Step}\ \mathit{down}, & \mathit{Step}\ \underline{\mathit{ruleRename}}, & \mathit{Step}\ \mathit{up}, & \mathit{Step}\ \mathit{down}, & \mathit{Step}\ \underline{\mathit{ruleRename}}, \\
\mathit{Step}\ \mathit{up}, & \mathit{Step}\ \mathit{not}, & \mathit{Exit}\ \ell_2, & \mathit{Enter}\ \ell_3, & \mathit{Step}\ \underline{\mathit{ruleAdd}}, \\
\mathit{Exit}\ \ell_3, & \mathit{Enter}\ \ell_4, & \mathit{Step}\ \underline{\mathit{ruleSimpl}}, & \mathit{Exit}\ \ell_4, & \mathit{Exit}\ \ell_0]
\end{array}
$$

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

4 Extensions

The previous section presents the core of our work on recognizing strategies: strategies can be labeled, and with the functions *empty* and *firsts* we can run a strategy. In this section we present three extensions to illustrate the flexibility of our approach.

4.1 Parallel strategies

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

In our setup, it is relatively easy to add a new constructor for parallel strategies to the *Grammar* data type, and we will further explore this approach.

$$\textbf{data}\ \mathit{Grammar}\ a = \ldots \mid \mathit{Grammar}\ a\ \mathrel{:||:}\ \mathit{Grammar}\ a$$

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

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

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

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

Other functions that operate on the *Grammar* data type (such as *freeVars* and *mapSymbol*) have to be extended as well, but these changes are minimal. Using a generic traversal library [10] can further reduce the impact of adding a constructor.

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

4.2 Removing left recursion

Because we can inspect the grammar, we can detect and remove left recursion in a grammar. Left-recursive definitions cause the function *firsts* to loop, and are therefore not desirable. Fortunately, removing left recursion from a context-free grammar is a standard procedure, and we can transfer this knowledge directly to our combinator approach. Consider the following left-recursive context-free grammar:

$$X \ ::= \ Xb \mid Xc \mid a \mid \epsilon$$

We proceed by grouping the left-recursive alternatives (Xb and Xc) and the remaining alternatives (a and ϵ), and arrive at the following grammar:

$$X \ ::= \ a \mid aY \mid \epsilon \mid Y \qquad\qquad Y \ ::= \ b \mid bY \mid c \mid cY$$

We briefly sketch how this procedure can be used for our *Grammar* data type. Given a *Rec* constructor, we want to make sure that none of its *Var*s appears first. We replace all variables of the *Rec* in question by a special symbol. Then, we use *firsts* to analyze the grammar, and we divide the alternatives in the left-recursive cases and the remaining cases. With the function *empty* we check for the ϵ alternative. In the last step, we combine all alternatives as we did for the context-free grammar. If necessary, we repeat the procedure until the left recursion has disappeared. Other existing grammar analyses can be reused in a similar way.

Example 4.1 We extend the smart constructors for recursive grammars (*rec* and *fix*) and let them check for left recursion. It is now safe to apply the function *firsts* to the left-recursive grammar *fix* ($\lambda x \rightarrow (x \mathbin{<\star>} \mathit{symbol}\ \texttt{'a'}) \mathbin{<|>} \mathit{succeed}$). Another example is *fix* ($\lambda x \rightarrow x$), which is simplified to *fail*.

4.3 Serializing the remaining strategy

In a recent project, we offered strategies as a service to the MathDox system [3]. For this binding, we designed a stateless protocol for diagnosing intermediate an-

swers submitted by students. One obstacle in establishing this binding was how to communicate the remaining strategy, which is part of the state of an exercise, back and forth. The representation of a strategy is finite and can be serialized. This is not very appealing because strategies can become quite large, which means that it takes longer to process a request.

The remainder of a strategy can also be encoded as a list of integers, with the extra benefit that the rewrite rules applied so far can be recovered. The encoding is rather simple: the integers in the list only encode which element of the *firsts* set has to be used. A *Prefix* associates symbols (rewrite rules) with the integers from the encoding, and contains the remaining grammar we are interested in:

$$\textbf{data}\ \mathit{Prefix}\ a = \mathit{Prefix}\ [(\mathit{Int}, a)]\ (\mathit{Grammar}\ a)$$

This data type is called *Prefix* because we are in the middle of a derivation, which means that we have a prefix of a sentence. The following function constructs a prefix from a list of integers and a labeled strategy:

$$\begin{aligned}
&\mathit{makePrefix} :: [\mathit{Int}] \to \mathit{LabeledStrategy}\ l\ a \to \mathit{Prefix}\ (\mathit{Step}\ l\ a)\\
&\mathit{makePrefix}\ \mathit{is} = f\ [\,]\ \mathit{is} \circ \mathit{withSteps}\\
&\quad \textbf{where}\ f\ \mathit{acc}\ [\,] \qquad s = \mathit{Prefix}\ (\mathit{reverse}\ \mathit{acc})\ s\\
&\qquad\quad f\ \mathit{acc}\ (i : \mathit{is})\ s = \textbf{case}\ \mathit{drop}\ i\ (\mathit{firsts}\ s)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad (a, t) : _ \to f\ ((i, a) : \mathit{acc})\ \mathit{is}\ t\\
&\qquad\qquad\qquad\qquad _ \qquad\quad \to \mathit{error}\ \texttt{"invalid prefix"}
\end{aligned}$$

The local function f has an accumulating argument which builds up a list in reverse order for efficiency reasons. This explains why the list *acc* has to be reversed in the case for the empty list. Each integer i from the list is used to select the ith element of the list returned by the function *firsts*.

Example 4.2 Let us compute the list of integers that encodes the full derivation of our running example (see Example 3.2):

$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$$

The list contains twenty elements, just like the list of steps. The list is dominated by zeros, which is not a coincident. At most places in our grammar, there is just one path that can be followed, witnessed by the fact that *firsts* returns a singleton list at these positions. A simple optimization is to not add an integer for the cases where *firsts* offers no choice.

5 Error diagnosis and hints

In this section, we explain how strategies can help in diagnosing student errors and reporting useful feedback and hints. We have implemented several kinds of feedback for our own exercise assistants, but also for other e-learning systems that use our feedback services as a back-end. The simplest form of feedback is correct/incorrect

for the final answer, which is the kind of feedback that is offered by several online tools. A useful extension to this categorization is to check for equivalence between the submitted term and the initial term. With this, we can distinguish a correct but not yet final answer from an incorrect answer. In the rest of this section we present a number of scenarios in the fraction domain to illustrate the possibilities.

Example 5.1 A student submits $\frac{16}{15}$ as the final answer. The exercise assistant reports back to the student that his answer is correct, but with a gentle reminder that the exercise is not yet finished. In this scenario, the strategy tells exactly which step needs to be done: the improper fraction should be simplified.

Example 5.2 A term is submitted as an intermediate step: the rule used is recognized, but according to the strategy it shouldn't have been applied. For instance, [Rename] is recognized, but the denominators of the fractions are already equal. The student can be warned that although the step is correct, it is better to do something else.

Worked-out problems can be generated from a strategy, showing all the steps to go from the initial term to the expected answer. A worked-out problem is the presentation of a sentence that is generated by the strategy. The next step in a derivation can be calculated with the function *firsts*. The information computed with *firsts* can be presented in different ways: the hints can be very general or very specific, for instance by using the levels of the labels in the strategy. A strategy can complete the exercise, and therefore, progress information, such as the number of steps remaining, is available.

Example 5.3 A student has no clue how to add the fractions $\frac{2}{5}$ and $\frac{2}{3}$, and presses the hint button. The system reports the hint: "make the denominators equal". The fact that the denominators are not yet equal can be concluded from the strategy. If this does not help the student, the system can emit a more specific message stating that the fractions should be renamed such that the denominators become 15. This number is calculated and present in the environment in which the strategy is run. A final hint could suggest to rewrite the part $\frac{2}{5}$ into $\frac{6}{15}$.

Strategies can work together with buggy rules: these rules capture common mistakes, and help to report specialized messages for specific (but often occurring) errors. In addition to the buggy rules, it is also possible to formulate buggy strategies, i.e., common procedural mistakes. In our fraction domain, for example, a buggy strategy would be to make the numerators equal before adding fractions.

Example 5.4 A student submits $\frac{4}{8}$ as the solution to the exercise $\frac{2}{5}+\frac{2}{3}$. Because the terms are not equivalent, the buggy rules are considered (B1 to B3 from Figure 1), and in this case, rule B1 matches. A special message associated with this rule (e.g., "it is not sound to add the numerators and the denominators of the fractions: rename the fractions first") is reported to the student.

Strategies and rules are essentially the same, except that the structure of a strategy is made explicit. Hence, it is straightforward to turn a strategy into a

rule, or a part of a strategy with a certain label. This is convenient if following a strategy becomes routine, and a step-wise approach is no longer helpful to the student. Similarly, a tool can ask a student to solve the entire problem first, and decompose the problem in steps if the submitted answer is not correct.

Example 5.5 A student is asked to provide the final answer to a question, and in case it is incorrect, the exercise tool poses sub-problems to the student. These sub-problems can be calculated automatically from the strategy by looking at the labels. The strategy for adding fractions, for instance, can be decomposed in 4 steps.

6 Related work

There are many tools that offer students an environment in which they can solve exercises incrementally, such as MathDox [3] and ActiveMath [5]. Most of these tools are limited to correct/incorrect feedback, because it is often difficult and laborious in these systems to diagnose mistakes. However, some tools use external domain reasoners for making a diagnosis, which is exactly what our strategy recognizer has to offer. Some work has been done on diagnosing student mistakes on the level of rewrite rules [2,7,15].

In this paper, we discuss the design and implementation of a strategy recognizer, which makes it possible to use strategies for improving error diagnosis. Strategies for specifying exercises are introduced in a different paper [6]. By viewing strategy recognition as a parsing problem, we take advantage of almost 50 years of experience in parsing sentences of context-free languages. The strategy language on which our work is based is very similar to languages that are used in parser libraries [9,13], but also to strategic programming languages such as Stratego [11,14] and Elan [1], data conversion libraries [4], and languages in other domains.

7 Conclusions

This paper presents a complete implementation of a recognizer for strategies. A strategy describes valid sequences of rewrite rules, and is very similar to context-free grammars. Knowledge and experience from the field of parsing sentences can be transferred to the domain of stepwise solving exercises. One example of such a transfer is the grammar transformation to remove left recursion.

Although it is tempting to reuse existing parsing tools and libraries, a closer look at the problem reveals subtle differences that make the existing tools unsuitable for dealing with the problem we are facing. Nevertheless, the strategy combinators that we selected as our starting point are inspired by context-free grammars. Some design choices were discussed, in particular how to deal with recursion, and how to mark positions in a strategy. In Section 5 we have shown how strategies can be used to report improved feedback.

We will continue our research on strategy recognizers in several directions. We are working on creating bindings with a number of existing tutoring tools, such as ActiveMath [5]. Protocols are needed to exchange information with such an

environment, and we are working on developing and standardizing these protocols. Our tool has a binding with MathDox [3], and has recently been used in a classroom setting. We have collected data from these session, and preliminary analyses show that providing feedback on the strategy level improves far transfer: students that received feedback on the strategic level did better in advanced exercises. A final area that requires further investigation is how to make strategies (and the associated feedback messages) more accessible to teachers.

References

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

[2] Bouwers, E., "Improving Automated Feedback – a Generic Rule-Feedback Generator," Master's thesis, Utrecht University, Department of Information and Computing Sciences (2007).

[3] Cohen, A., H. Cuypers, E. Reinaldo Barreiro and H. Sterk, *Interactive mathematical documents on the web*, in: *Algebra, Geometry and Software Systems* (2003), pp. 289–306.

[4] Cunha, A. and J. Visser, *Strongly typed rewriting for coupled software transformation*, Electronic Notes in Theoretical Computer Science **174** (2007), pp. 17–34.

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

[6] Heeren, B., J. Jeuring, A. v. Leeuwen and A. Gerdes, *Specifying strategies for exercises*, in: S. Autexier et al., editor, *Proceedings of MKM 2008*, LNAI **5144** (2008), pp. 430–445.

[7] Hennecke, M., "Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen (in German)," Ph.D. thesis, Hildesheim University (1999), fortschritt-Berichte VDI Reihe 10, Informatik / Kommunikationstechnik; 605. Düsseldorf: VDI-Verlag.

[8] Huet, G., *Functional Pearl: The Zipper*, Journal of Functional Programming **7** (1997), pp. 549–554.

[9] Hutton, G., *Higher-order Functions for Parsing*, Journal of Functional Programming **2** (1992), pp. 323–343.

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

[11] Lämmel, R., E. Visser and J. Visser, *The Essence of Strategic Programming* (2003), 20 p.; Draft as of October 8, 2003; Available from http://homepages.cwi.nl/~ralf/eosp/paper.pdf.

[12] Peyton Jones et al., S., "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/.

[13] Swierstra, S. D. and L. Duponcheel, *Deterministic, error-correcting combinator parsers*, in: J. Launchbury, E. Meijer and T. Sheard, editors, *Advanced Functional Programming*, LNCS **1129** (1996), pp. 184–207.

[14] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *ICFP '98*, 1998, pp. 13–26.

[15] Zinn, C., *Supporting tutorial feedback to student help requests and errors in symbolic differentiation*, in: M. Ikeda, K. Ashley and T.-W. Chan, editors, *ITS 2006*, LNCS **4053** (2006), pp. 349–359.