Universiteit Utrecht

Center for Software Technology
Universiteit Utrecht
The Netherlands

Thesis for the degree of Master of Science

INF/SCR-07-013

Improving Automated Feedback
Building a Generic Rule-Feedback Generator

Author:
Eric Bouwers
embouwer@cs.uu.nl

Daily Supervisor:
Prof. Dr. Johan Jeuring
johanj@cs.uu.nl

Second Supervisor:
Dr. Jurriaan Hage
jur@cs.uu.nl

September, 2007

Abstract

The importance of feedback in learning a procedural skill cannot be ignored. It is therefore surprising to see that many of the existing educational tools aimed at learning these skills fail to provide feedback on a level above *this is incorrect*. Those tools that do manage to provide higher quality feedback are the result of several years of research. Transferring this research to different domains is not interesting research material, and therefore not done by the academic world. On the other hand, commercial implementers do not seem to want to spend large amounts of resources on improving the feedback part of their software. This is probably because the trade-off between investing in research and the expected profit is of no commercial interest.

We aim at solving this problem by combining both existing as well as new research into one generic framework. Instantiating this framework on a particular domain results in a configurable tool that can provide feedback after each step in solving an exercise. The quality of this feedback ranges from a simple *incorrect* to a custom-made feedback message, depending on the level and amount of configuration that is used. As an example, we instantiate the framework on three different domains and evaluate the feedback with several domain-specialists. The result is a solid framework that can serve as a basis for future research as well as a prototype for further development.

Contents

I Goal

1 Introduction **1**

2 Context **3**
- 2.1 Concepts 3
- 2.2 Feedback in current systems 4
- 2.3 User interface 13
- 2.4 Summary 14

3 Research question and approach **15**
- 3.1 Research approach 15
- 3.2 Instantiation 16
- 3.3 Generation of rule-feedback 16
- 3.4 Limitations 17
- 3.5 Summary 18

II Design

4 Rule Feedback Generator **21**
- 4.1 Overall design 23
- 4.2 Instantiation 23
- 4.3 Phase 1 24
- 4.4 Phase 2 25
- 4.5 Phase 3 25
- 4.6 Phase 4 35
- 4.7 Phase Collaboration 37
- 4.8 Summary 38

5 Graphical User Interface **39**
- 5.1 Screens 39
- 5.2 Summary 43

III Implementation

6 Rule Feedback Generator **47**
6.1 Type class for rewriting . . . 47
6.2 Instantiating a domain . . . 49
6.3 Phase 0 . . . 51
6.4 General structure . . . 51
6.5 Summary . . . 52

7 Graphical User Interface **53**
7.1 Client Side . . . 54
7.2 Server Side . . . 55
7.3 Conclusion . . . 55

IV Evaluation

8 Evaluation **59**
8.1 Test suite evaluation . . . 59
8.2 Logic . . . 61
8.3 Real world tests . . . 63
8.4 Conclusion . . . 65

9 Future Work **67**
9.1 Third phase algorithm . . . 67
9.2 The Framework . . . 68
9.3 The GUI . . . 68

10 Conclusion **69**

V Appendixes

A Rewrite rules **73**
A.1 Syntax and semantics . . . 73
A.2 Checks . . . 74

Bibliography **74**

Index **77**

List of Figures

2.1 Abstract representation of $\frac{1}{2} + \frac{1}{4}$ 4
2.2 Tree representation of $\frac{1}{2} + \frac{1}{4}$ 4
2.3 solve_mul_frac : Strategy for multiplying two fractions 5
2.4 Math(X)Pert main screen 6
2.5 Hint in Math(X)pert 6
2.6 Eight design principles of Beeson 7
2.7 Aplusix main screen 8
2.8 Hint in Aplusix 8
2.9 Feedback in Freudenthal applets 9
2.10 The Equation solver 9
2.11 LeActiveMath exercise screen 10
2.12 Bugfix feedback screen. 11

4.1 Main components of the system 22
4.2 Main design of the RFG 23
4.3 Code for instantiation 24
4.4 Code for phase 1 25
4.5 Code for phase 2 25
4.6 Code for phase 3 26
4.7 AST representation of $1 + 4 + 7, 7 + 3$ and the resulting rewrite rule 29
4.8 Code for general recursion. 32
4.9 Code for phase 4 35

5.1 Initial screen 41
5.2 Screen with merged view 41
5.3 Screen with rule 42
5.4 Screen with error screen 43

6.1 RFG type class 48
6.2 Logic data-type 50
6.3 RFG Expr instance 50

7.1 Components of the GUI 54

List of Figures

A.1 Syntax of rules in BNF 74

List of Tables

2.1 Different classes of rule-feedback generation 13

3.1 Different phases in rule-feedback generation 18

4.1 Available command line options 22

6.1 List of modules of the framework 52

8.1 Allowed rules for fractions 60
8.2 Test scenarios for fractions 60
8.3 Allowed rules for logic 62
8.4 Test scenarios for logic 62
8.5 Allowed rules for equations 64
8.6 Test scenarios for equations 64

Part I

Goal

Chapter 1

Introduction

In order to graduate from any level of education a student has to master some skills. Acquiring any type of skill generally involves some kind of feedback. Positive feedback lets the student know that the right track is being followed, negative feedback (hopefully) guides the student back to it. Research on feedback in education has shown that immediate feedback is a critical aspect of successful learning [12, 28].

Procedural skills are often used in mathematics and computer science to solve a problem in a general way. An example of this is solving a system of linear equations by subtracting equations from top to bottom, and then substituting variables from bottom to top. Procedural skills are usually well-defined for domains that manipulate an expression according to a set of rewrite rules.

In today's classroom a procedural skill is usually taught by means of examples presented by the teacher. After the explanation of the procedure, which may or may not be guided by the ideas of the students themselves, each student practices the skill by solving exercises on paper. Since it is physically impossible for a teacher to give immediate, semantically rich and personal feedback to every student in a normal classroom-setting, feedback is delayed. It is not uncommon that feedback is given after a period of hours or even days.

A partial solution to this problem is the usage of e-learning systems to practice a particular procedure. Procedures that can be practiced with these systems include driving a car and flying a plane, but we only target typical classroom skills in the area of mathematics and logic.

E-learning systems offer the possibility to generate immediate feedback after every step (*rule-feedback*), or a series of steps (*strategy-feedback*). Furthermore, such a system can collect knowledge about each student and use this knowledge to generate exercises and feedback that matches the knowledge level of the student. By analyzing the information of a group of students it can also identify common misconceptions. These systems cannot replace a teacher completely, but when they use the right feedback they can certainly be a useful addition to the educational program [25].

There are many different e-learning systems available that can be used to practice procedural skills. These systems vary widely in educational motivation, user-interface, breadth, width, etc. One thing that almost all of these systems have in common is, unfortunately, the lack of sophisticated feedback. The general form of feedback is often simply *correct* or *incorrect*, sometimes in a more attractive wording. When a more sophisticated feedback mechanism exists it is often hard coded in the system, or even in the exercises. This limits the usability of these systems because they just replace the exercises on paper. Such a simple replacement does not add a lot to the learning process and can even increase the workload for teachers.

This thesis investigates techniques that can be used to add sophisticated, domain-specific rule-feedback to new as well as to existing tools. By combining techniques from the field of term-rewriting, strategies and generic programming we supply a generic framework that supports the generation of domain-specific rule-feedback with minimal effort. This (partially) answers the second research question posed by Jeuring and Passier [29]:

> How can we specify domain-specific feedback?

Chapter 2

Context

In order to illustrate the current problem with feedback in education, and to be able to define the goal of this thesis we need to give the context. Therefore, this chapter describes the concepts which are needed to generate domain-specific feedback in Section 2.1. The current status of feedback in educational tools are discussed in Section 2.2. Several guidelines for building a Graphical User Interface (GUI) for educational tools are mentioned in Section 2.3.

2.1 Concepts

In order to generate domain-specific feedback a teacher needs to "understand" the domain[1]. This tenet, cited as early as 1985 by Bundy [6], implies that intelligence has to be modeled in e-learning programs. Bundy has identified three needs that always arise when dealing with this task:

1. The need to have knowledge about the domain.
2. The need to reason with that knowledge.
3. The need for knowledge about how to direct or guide that reasoning.

We describe a domain, for example the domain of fractions, by an *abstract syntax* of its expressions and the operations that are permitted on these expressions. An example of an abstract representation of the addition of two fractions is shown in Figure 2.1. This representation of expressions is not intuitive and not useful for presentation purposes. So an expression also needs a *concrete representation* that can be used in a (G)UI, for example the standard representation $\frac{1}{2} + \frac{1}{4}$.

The abstract syntax of an expression can also be seen as a tree, see Figure 2.2. This tree representation of an expression allows us to model the operations on an expression as operations on nodes in a tree. These operations can be defined by *rewrite rules*.

[1]This holds for both computers and human teachers

```
Add(Fraction(Num(1)
            ,Num(2)
            )
   ,Fraction(Num(1)
            ,Num(4)
            )
   )
```

Figure 2.1: Abstract representation of $\frac{1}{2} + \frac{1}{4}$

```
      Add
     /   \
   Frac   Frac
  /   \   /  \
Num Num Num Num
 |   |   |   |
 1   2   1   4
```

Figure 2.2: Tree representation of $\frac{1}{2} + \frac{1}{4}$

An example of a rewrite rule is given by the rule `comm_add` which describes commutativity of addition:

`comm_add`: $A + B \rightarrow B + A$

A rewrite rule consists of a name, a matching expression (the left-hand side of the arrow), and a build expression (the right hand side of an arrow). Rewrite rules can use meta-variables, like `A` and `B` in `comm_add`, to match on sub-expressions. So the `comm_add` rule can be used to rewrite the expression $\frac{1}{2} + \frac{1}{4}$ to $\frac{1}{4} + \frac{1}{2}$ by matching A to $\frac{1}{4}$, and B to $\frac{1}{2}$.

A distinction is made between rules without restrictions, and rules with restrictions. The above rewrite rule falls under the first category, an example of the second category is:

`solve_mul`: $A * B \rightarrow C$ `where` $C := \mathit{solve}(A * B)$

This rule solves the multiplication of two terms. The keyword `solve` in the rule stand for the evaluation of the operator, in this case the multiplication.

Rewrite rules like `solve_mul` can be used in a sequence of rewrite rules to form a *strategy*. Strategies express a procedural skill in a formal way. A strategy for the procedural skill of multiplying a fraction is shown in Figure 2.3 . Both terms and rewrite rules are defined in concrete syntax.

Strategies can be used by other strategies in the same way that procedural skills may depend on other procedural skills. For example, adding two fractions with different denominators depends on the multiplication of two fractions.

2.2 Feedback in current systems

Automatically generating feedback is not a new problem in computer science. Many approaches have been investigated in order to address this problem, but none of them seems to solve the problem in an easy and generic manner.

Beeson has taken drastic measures to solve this problem in Math(X)Pert [1, 2]. This tool allows a student to select a (part of) the current term and then shows the rewrite rules that can be applied to this selection, see Figure 2.4.

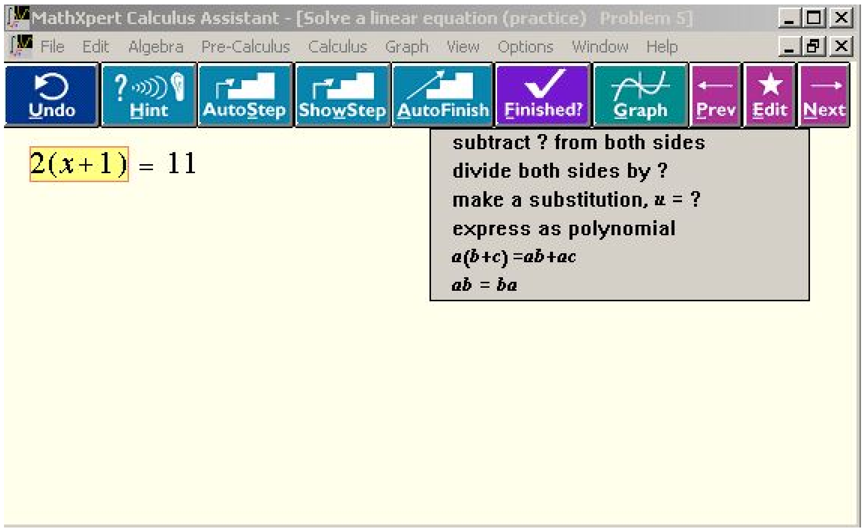
Current Term	Step
$\frac{x_1}{y_1} * \frac{x_2}{y_2}$	
	$\downarrow$ `(A/B) * (C/D)`$\rightarrow$ `(A*C)/(B*D) : mul_fracs`
$\frac{(x_1 * x_2)}{(y_1 * y_2)}$	
	$\downarrow$ `eval_mul` (numerator)
$\frac{solve(x_1 * x_2)}{(y_1 * y_2)}$	
	$\downarrow$ `eval_num` (denominator)
$\frac{solve(x_1 * x_2)}{solve(y_1 * y_2)}$	

Figure 2.3: solve_mul_frac : Strategy for multiplying two fractions

After a selection is made Math(X)Pert automatically applies the rule, which eliminates the need for rule-feedback. This approach allows a student to focus on the procedural skill for solving an exercise. Strategy-feedback is not given to a student explicitly, but a student can ask for a hint or the next step to take; an example of this is shown in Figure 2.5. However, there is also some implicit strategy-feedback given to the student when a term is selected. The program only shows the rewrite rules that can be applied to the selection and thus narrows down the choices for a student.

The decision for this approach is based on a pedagogical vision, in this case the vision of Maria Montessori. This shows that the choice for a particular pedagogical vision heavily influences the design and feedback possibilities of a tool, something which is also confirmed by Zuidema *et al.* [35]. But the design of Math(X)Pert was not only based on a pedagogical vision. Beeson has also listed eight principles that need to be met in order to provide successful computer support for education in algebra, trigonometry and calculus. These principles are listed in Figure 2.6 with a short explanation where necessary. A more detailed explanation of these principles can be found elsewhere [2].

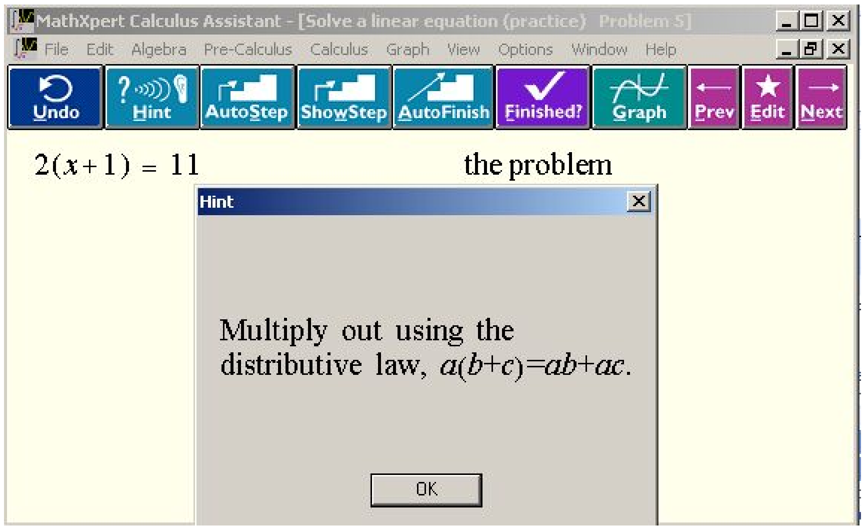
While Math(X)Pert restricts the student in the sense that a student cannot apply a rewrite rule incorrectly, tools like Aplusix [9] allow the complete manipulation of a term. A student starts with a term and is asked to construct a rewritten term from scratch. The tool does not show the rules that can be applied to the term. The strategy-feedback in Aplusix consists of two indicators on a status bar. The first one indicates whether a newly constructed term is semantically equivalent to the previous term. The other indicates if the rewrite step that is taken is a significant step in the solving process. A screen shot of the main screen of Aplusix is shown in Figure 2.7 This last indicator shows another difference between this tool and Math(X)Pert. A student who uses Aplusix can use the indicator to see whether certain steps lead towards a solution, whereas Math(X)Pert students can wander around in circles without making progress. Rule-feedback in Aplusix consists of a red-colored, crossed-out double-arrow when the new term is not a rewrite of the old term. This double-arrow is black when the step is correct. Aplusix does not stop a student when a rewrite step was applied incorrectly.


MathXpert Calculus Assistant - [Solve a linear equation (practice) Problem 5]
File Edit Algebra Pre-Calculus Calculus Graph View Options Window Help
Undo Hint AutoStep ShowStep AutoFinish Finished? Graph Prev Edit Next

$2(x+1) = 11$

subtract ? from both sides
divide both sides by ?
make a substitution, u = ?
express as polynomial
$a(b+c) = ab+ac$
$ab = ba$

Figure 2.4: Math(X)Pert main screen


MathXpert Calculus Assistant - [Solve a linear equation (practice) Problem 5]
File Edit Algebra Pre-Calculus Calculus Graph View Options Window Help
Undo Hint AutoStep ShowStep AutoFinish Finished? Graph Prev Edit Next
Hint
OK

$2(x+1) = 11$

the problem

Multiply out using the distributive law, $a(b+c)=ab+ac$.

Figure 2.5: Hint in Math(X)pert

1. Cognitive fidelity (solve the exercise like the student would)
2. Glass box (show how the solution can be solved)
3. Customized to the level of the user
4. Correctness (e.g. take conditions into account)
5. User is in control
6. The computer can take over if the user is lost
7. Easy of use
8. Usable with standard curriculum

Figure 2.6: Eight design principles of Beeson

When a student checks a set of rewrite rules that contains a faulty rewrite step, the system responds with messages similar to the one shown in Figure 2.8.

The Freudenthal Institute for Science and Mathematics Education has developed a set of Java-applets [15] which offer functionality similar to Aplusix. The applet for solving linear equations generates a term which the student must rewrite and simplify. A new term cannot be entered before the student has defined the exact rewrite rule or has chosen to use the 'equal to' step. When a student enters the rewritten term the system checks whether the new term is valid *according to the given rewrite rule*. The rule-feedback that is generated consists of either a red cross, see Figure 2.9 , or a green curl. In contrast with Aplusix, a red cross in the applets do stop a student from taking a new step. There is no notion of strategy-feedback within the applets.

Where Math(X)Pert eliminates the need for rule-feedback, both the Freudenthal applets and Aplusix just informs the student that they made a mistake. The feedback does not help a student to figure out what the source of the mistake was. Students who receive this type of feedback can become frustrated or even angry. This can be prevented by including some kind of hint in the feedback to help the student in understanding the mistake.

A joint venture between the Open Universiteit and Universiteit Utrecht has resulted in two exercise assistants that aim at providing rule-feedback with informative hints in an interactive fashion [23, 30]. Both of these tools are designed to mimic the traditional pen-and-paper approach (something which is also explicitly done by Raymond [31]) by asking the student to rewrite the current term in a text field, and generating feedback based upon the current and previous term. A screen shot is shown Figure 2.10. These tools are currently not able to give strategy-feedback, but this is part of ongoing research. The rule-feedback is generated *without* the need to specify the rule that is applied. This unique feature produces good results, but is currently unable to handle more than one rewrite step at the time.

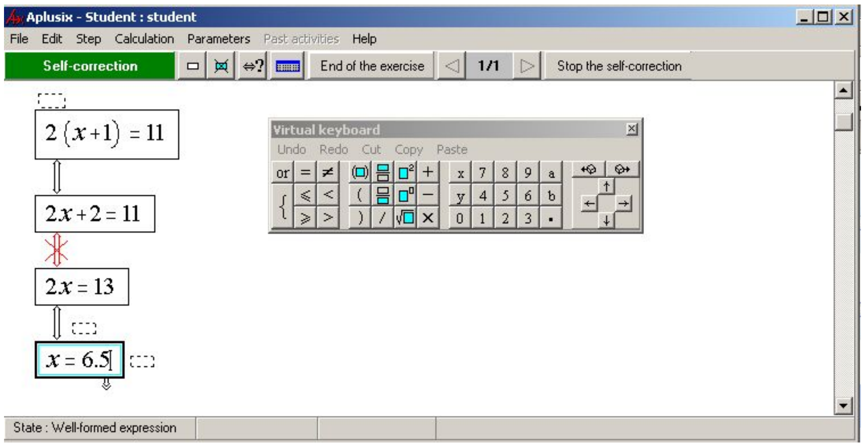
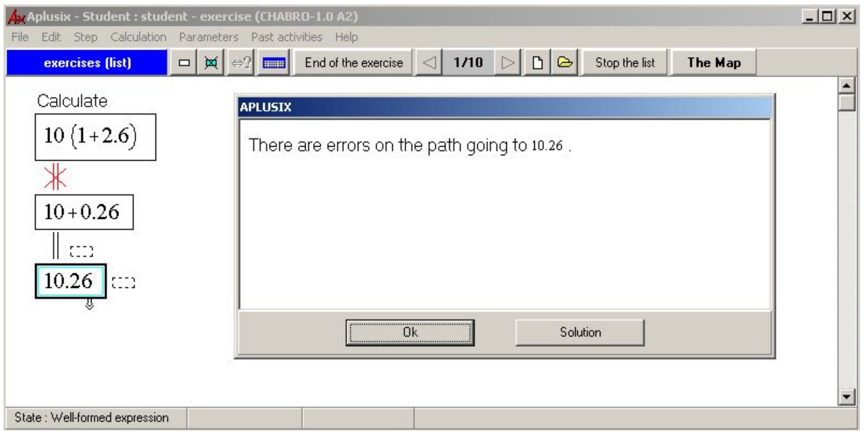


Figure 2.7: Aplusix main screen


Aplusix - Student : student - exercise (CHABRO-1.0 A2)
File Edit Step Calculation Parameters Past activities Help
exercises (list)
End of the exercise
1/10
Stop the list
The Map
Calculate
$10(1+2.6)$
$10+0.26$
10.26
APLUSIX
Ok
Solution
State : Well-formed expression

There are errors on the path going to 10.26 .

Figure 2.8: Hint in Aplusix

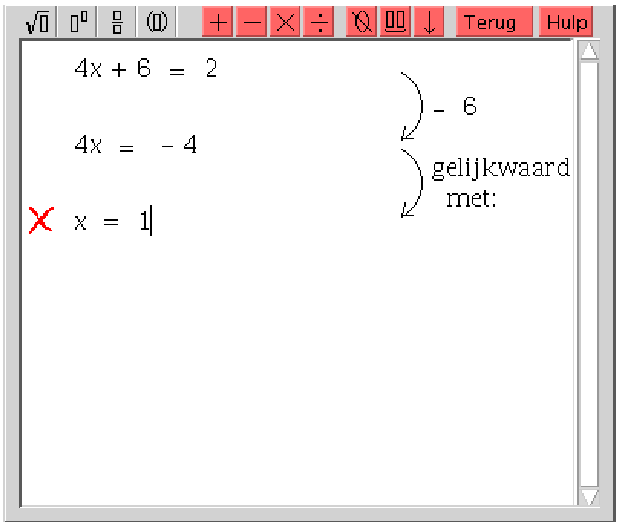
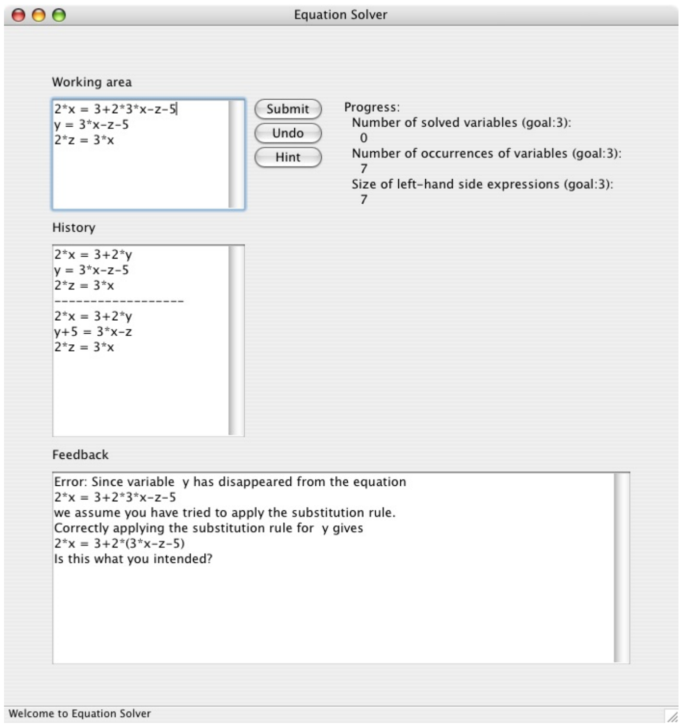


Figure 2.9: Feedback in Freudenthal applets



Working area

```
2*x = 3+2*3*x-z-5
y = 3*x-z-5
2*z = 3*x
```

Progress:

```
  Number of solved variables (goal:3):
    0
  Number of occurrences of variables (goal:3):
    7
  Size of left-hand side expressions (goal:3):
    7
```

History

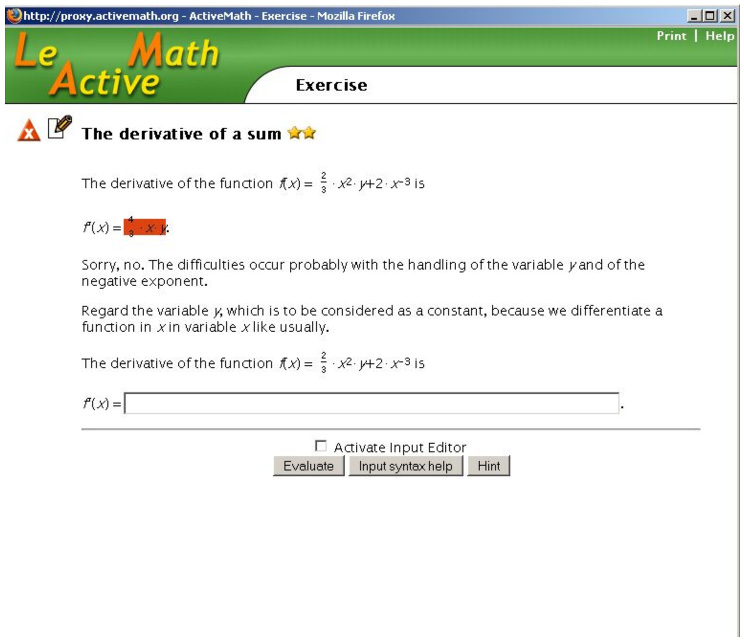
```
2*x = 3+2*y
y = 3*x-z-5
2*z = 3*x
------------------
2*x = 3+2*y
y+5 = 3*x-z
2*z = 3*x
```

Feedback

```
Error: Since variable  y has disappeared from the equation
2*x = 3+2*3*x-z-5
we assume you have tried to apply the substitution rule.
Correctly applying the substitution rule for  y gives
2*x = 3+2*(3*x-z-5)
Is this what you intended?
```

Welcome to Equation Solver

Figure 2.10: The Equation solver


http://proxy.activemath.org - ActiveMath - Exercise - Mozilla Firefox
Print | Help
LeActiveMath
Exercise

The derivative of a sum

The derivative of the function $f(x) = \frac{2}{3} \cdot x^2 \cdot y + 2 \cdot x^{-3}$ is

$f'(x) = \frac{4}{3} \cdot x \cdot y$.

Sorry, no. The difficulties occur probably with the handling of the variable y and of the negative exponent.

Regard the variable y, which is to be considered as a constant, because we differentiate a function in x in variable x like usually.

The derivative of the function $f(x) = \frac{2}{3} \cdot x^2 \cdot y + 2 \cdot x^{-3}$ is

$f'(x) =$ ______ .

☐ Activate Input Editor

Evaluate | Input syntax help | Hint

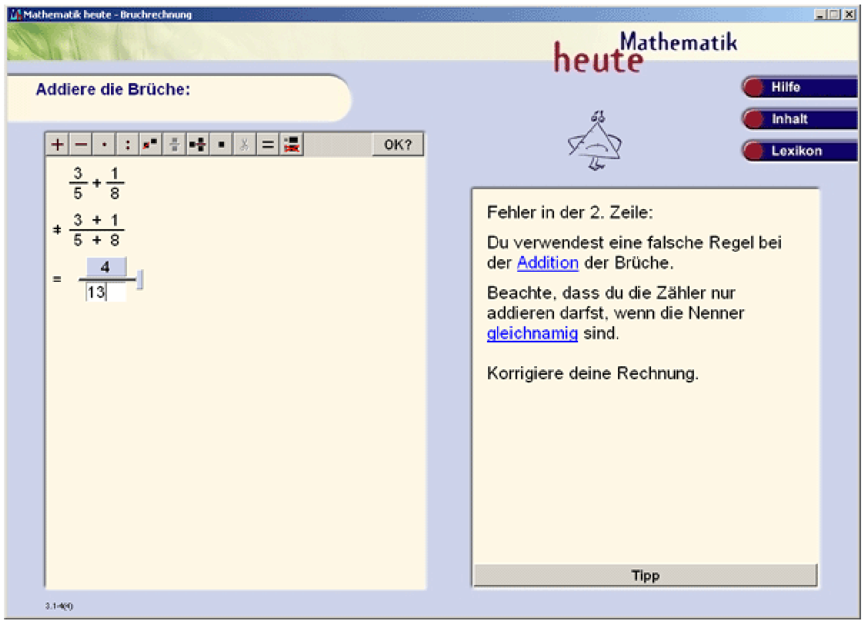
Figure 2.11: LeActiveMath exercise screen

ActiveMath [11, 17] tries to provide hints by allowing teachers to specify feedback for a specific class of exercises. The implementer of the specification needs to specify which subproblems there are within a class, together with feedback for each subproblem. ActiveMath can handle any instantiation of this class of exercises when this is done. Because feedback is specified for a specific class of exercises, the feedback can contain references to the domain of this class. The downside of this approach is that feedback is specified in a generic way, so the feedback that is generated is also generic. This leads to feedback similar to the following, received after giving $\frac{4}{3}xy$ as answer to the derivation of $\frac{2}{3}x^2y + 3x^{-3}$ with respect to x:

> Sorry, no. The difficulties occur probably with the handling of the variable y and of the negative exponent.

The original context of this error-message is shown in Figure 2.11. Even though this feedback is more informative then a simple *incorrect*, a student can still become frustrated when this message shows up several times on different terms.

So defining either strategy-feedback or rule-feedback in a general specification without having access to the current exercise leads to unwanted generalizations. There exist many tools, for example [3, 24, 33] and many multiple-choice programs, that try to deal with this problem by specifying the rule-feedback together with each individual exercise. However, the specification of this feedback is tedious, labor-intensive and sometimes not even straightforward. A balance between generic and individual feedback specification needs to be made.



$\frac{3}{5} + \frac{1}{8}$

$= \frac{3+1}{5+8}$

$= \frac{4}{13}$

Fehler in der 2. Zeile:

Du verwendest eine falsche Regel bei der Addition der Brüche.

Beachte, dass du die Zähler nur addieren darfst, wenn die Nenner gleichnamig sind.

Korrigiere deine Rechnung.

Figure 2.12: Bugfix feedback screen.

This balance can be achieved by using so called *buggy rules* to accurately identify the mistake a student made. An example of such a rule is:

`frac_add_faulty : A/B + C/D` $\longrightarrow$ `(A+C)/(B+D)  where B != D`

Research on finding sets of buggy-rules dates from 1980, with BUGGY [7, 5] and DEBUGGY [4], up until 2003 with BUGFIX [18]. The older tools were specifically designed to practice the diagnosis of errors by teachers. BUGGY presents the user a set of problems together with the (faulty) answers. The task is then to diagnose and describe the misconception of the virtual student. Although the research mentions an experiment in which students use this tool, the focus was on the diagnoses and not on the generation of (rule-)feedback. The opposite is true for BUGFIX which is used in a commercial product, Mathematik Heute, to generate rule-feedback based on buggy rules.

A screen shot of the main screen of Mathematik Heute is shown in Figure 2.12. The error message on the left can be translated to:

> Error in the second step:
> You are using an incorrect rule to add two fractions
> Keep in mind that you can only add the numerators when the denominators are equal.
> Correct your calculation.

This message is generated by BUGFIX after the calculation of possible paths from the current term back to the old term. By using normal as well as buggy rules BUGFIX models the error that is probably made by the student. This allows the generation of semantically rich feedback messages that help the student to understand the mistake that has been made.

The main problem with this approach, already mentioned by Beeson [1], is the extensive work that has to be done to gather a representative set of buggy-rules. Hennecke [18] reports 350 rules for the domain of fractions with adding, subtracting, multiplication, and division. Using a representative set of buggy- and normal rules as a basis for a tool can lead to a development process of several years. But the results of this technique are very promising, because it allows the generic specification of common mistakes without losing the ability to refer to the current term.

In order to summarize the current status of generated rule-feedback, we distinguish several classes of tools[2]. A list of these different classes, together with examples of tools belonging to them, is shown in Table 2.1. The following list explains the phases in a more elaborate manner.

- The first class of tools only informs a student whether a rule was applied correctly by inspecting the answers of the previous term and the current term. If the answers do not match the student is told that the rule is not correctly applied, resulting in rule-feedback that only contains correct/incorrect. This kind of feedback is valid, but it does not help a student to identify the mistake that has been made.

- A second class of tools generates more informative rule-feedback by using a set of allowed rules to "guess" the applied rule. The rule-feedback that is generated by this class contains a hint about how to correctly apply the rule. This feedback can be used by the student to find out what went wrong with the application of the rule.

- The third class of tools differs from the second class by not only using the normal rules, but also a set of buggy rules to determine the mistake a student has made. The rule-feedback generated by this class of tools states the mistake a student has made and information about the correct rule which can be applied.

- Two types of tools are combined within the fourth class of tools. This class consists of the tools that specify rule-feedback generically for a set of problems or on a per exercise basis. This results in either rule-feedback which is too generic or too restrictive to be useful for a student.

- The last class of tools simply eliminates the need for rule-feedback by applying a rule automatically.

[2]Notice that this categorization is probably different for the generation of strategy-feedback.

Class name	Description	Tool
Correct/Incorrect (CI)	Rule-feedback just indicates whether the application of a rule was correct.	Aplusix, Freudenthal applets
Normal Rules (NR)	Generate rule-feedback with hints based on the normal set of allowed rewrite-rules.	IDEA-tools
Buggy Rules (BR)	Use normal- as well as buggy-rules to generate rule-feedback with hints.	BUGFIX
Generic/Individual (GI)	Specify rule-feedback for a class of exercises or on a per-exercise basis.	ActiveMath, Stack
No Feedback (NF)	Eliminate the need for rule-feedback by applying rules automatically.	Math(X)Pert

Table 2.1: Different classes of rule-feedback generation

2.3 User interface

In conventional software a tool can be used because the user is familiar with the content domain. An example of this is the usage of a word-processor. A user is often familiar with methods of text manipulation so the interface can be deciphered by using the knowledge about the domain. However, educational tools aim at learning the domain, hence the interface must be intuitive and easy to use without relying on information from the domain. The following quote from Frye [16] summarizes the importance of a good interface:

> With educational software, the interface must provide an entry to the content domain rather than vice versa.

Although there has been extensive research on the design of interfaces in general, see Schneiderman [34] for an extensive overview, the specific research on interfaces for educational tools has been limited. However, there are a few guidelines that can be defined by examining the interface of current tools, the general research and some specific research ([19, 35]). These guidelines are quite basic and may seem superfluous, but almost all of the tools discussed in Section 2.2 violate one or more of these guidelines.

First, the student solves an exercise from top to bottom, the direction in which a student generally reads. Second, when a student is allowed to enter an expression it can use either a textual or a more visual approach. In the first case a student just types `1/2` in a box. In the second case, a button with a picture like $\frac{*}{*}$ is chosen after which a 1 and a 2 is typed. A property of the last input-type must be that the generated expression is always accepted by the system. To be more useful, the textual representation has to support different notations of the same construction. For example both `2.x` and `2x` encode the two times `x` construct. Third, any action of the student must be followed by a response in a very short period of time (≤ 2 seconds). Otherwise the student is slowed down and becomes agitated or bored. Fourth, if there is rule-feedback, it is generated visually close to the term with the applied rule. This guarantees that the student does not overlook the feedback and is aware of the possible hints before taking the next step.

2.4 Summary

In order to generate domain-specific feedback we need to model the domain in terms a computer can understand. We have shown how this can be done by using abstract syntax trees and rewrite rules. Furthermore, we have examined some of the current educational tools and have distinguished several classes of feedback that these tools provide. The main observation is that the quality of feedback increases when more information is given to the tools. Finally, we have stated four guidelines for the design of a GUI for educational tools. Although these guidelines seem obvious, they are frequently violated by current tools.

Chapter 3

Research question and approach

Tools like BUGFIX show that it is possible to generate high-quality rule-feedback containing hints about the (possible) mistake of a student. It is therefore surprising to see that, considering the importance of feedback in education, most of today's tools are generating feedback on the level of correct/incorrect. We think that this gap between research and implementation exists because of the following reasons:

1. Developers of educational tools are unaware of the possibilities.
2. The amount of effort needed to implement a high-quality rule-feedback generator for a specific domain, let alone different domains, is too high.
3. There is no simple approach to take a "road in the middle". Each method of rule-feedback generation is to be implemented completely, or not at all.

The last two problems lead to rigid implementations which cannot be adapted to fit the needs of a specific class-room. This is unfortunate because research shows that adaptivity is of great value to teachers [8, 19, 26].

The goal of this thesis is to improve the quality of automatically generated rule-feedback in a generic fashion. More formally, our research question is:

> How can we make the generation of high-quality, domain-specific feedback easier?

3.1 Research approach

We aim at providing a *generic framework* that separates the implementation of the techniques for rule-feedback generation from knowledge about a domain. This reduces the effort to implement high-quality rule-feedback generators to the specification of the domain. The repetitive tasks of implementing specific generation techniques is being handled by the framework. In order to provide easy access to other producers, the framework becomes available with precise documentation. After instantiation of the framework the generated command-line tool can be configured in an *incremental* fashion, i.e. there is no need to specify all of the rules for a complete domain at once. This allows a partial implementation of methods which makes the tool adaptive to the level of a single class-room. The best rule-feedback is generated with a full configuration, but rule-feedback above the level of correct/incorrect can already be obtained with a partial configuration file.

Within the remainder of this chapter we explain the desired instantiation of the framework. Additionally, we give a high-level overview of the different phases in which feedback is generated in combination with expected input. Also, we discuss some limitations of this approach.

3.2 Instantiation

Instantiating the framework on a domain is achieved by supplying a *parser* module and a *solver* module, both domain-specific. The parser must be able to parse a string-representation of a term in the domain to the abstract syntax. This parser is also used by the framework to parse the rewrite rules that can be supplied. The solver must be able to calculate an answer given a term. This answer is used as the result that the student must supply to finish the exercise.

Apart from the obligatory parser and solver, the framework also expects an optional domain-specific *equality* module. When this module is supplied the resulting tool uses it to check two terms for equality. Supplying this module prevents situations in which, for example, the answer $1\frac{1}{2}$ is rejected as an answer to the exercise $\frac{1}{2} + \frac{1}{2} + \frac{1}{2}$, because the solver only accepts $\frac{3}{2}$.

3.3 Generation of rule-feedback

After instantiating the framework for a domain the resulting tool generates rule-feedback in different phases. These phases combine the best elements of the methods which were listed in Section 2.2. A more advanced phase generates more informative and more accurate feedback, but requires more input. The phases are introduced by an example in which a student is performing an exercise to add fractions. The exercise starts with the term $\frac{4}{5} + \frac{3}{10}$ and the student has rewritten this into $\frac{4+3}{5+10}$.

The default level of feedback that the student received is defined in phase 1. This phase only needs the current term (CT), in this case $\frac{4+3}{5+10}$, and the previous term (PT), in this case $\frac{4}{5} + \frac{3}{10}$, to generate feedback. The tool uses the solver to calculate the result of both terms, i.e. the answer the student must give in order to solve the problem represented by the term. When the results are the same, the tool generates a *correct*-message, otherwise an *incorrect*-message. The quality of feedback generated in this phase can be compared to the level of feedback in Aplusix and the Freudenthal Institute: it is easy to specify, but it does not help a student to identify her mistake.

Phase 2 needs some more information in order to generate better feedback. The student has to define the rule she uses prior to the actual application of the rule. In the example the student probably supplies a rule similar to:

`Student_rule` : $\frac{a}{b} + \frac{c}{d} \rightarrow \frac{a+c}{b+d}$

The system checks whether the given rule is applied correctly, but also falls back to phase 1 before generating a message. In this case the message will inform the student that the rule was applied correctly, but that the answer of the exercise has changed. This information helps the student to identify the supplied rule as the source of the mistake instead of the application of the rule. The actual mistake is still unknown, but the search-space is smaller. The feedback in this phase is easy to specify and more informative than the feedback of phase 1. Unfortunately, the burden of specifying rules is put on the student who is learning the rules.

Phase 3 lifts the burden of specifying rules from the user by accepting a set of allowed rules as configuration. This shifts the responsibility from the student to the implementer or the teacher. In this example the set contains the rule (amongst others):

`add_frac` : $\frac{a}{b} + \frac{c}{b} \rightarrow \frac{a+c}{b}$

Within the third phase the tool figures out that this is (probably) the rule the user intended to use. The message that is generated contains references to this rule and can explain why the rule cannot be applied. The definition of the rules by the implementer takes some time, but the feedback can be generated for all the problems within the domain. This is the same level of generality offered by ActiveMath, but the quality of the feedback our tool generates is higher because it involves the actual current term.

The problem with the feedback from phase 3 is that it "guesses" the rule that the student wanted to apply. This can result in feedback that is not useful because the guess was wrong. Phase 4 partially solves this problem by accepting an additional set of buggy-rules. These rules contain common mistakes, for example the rule entered by the student in phase 2. When the student uses one of these buggy-rules in an exercise the tool "knows" the mistake. This knowledge can be used to generate a very specific message, in the example this can be *"You cannot add fractions with different denominators."*. By defining a custom feedback message with a particular buggy-rule, the feedback can even be adapted to the situation of a particular class-room.

An overview of the four phases is given in Table 3.1. Notice that the higher phases do not assume that all possible rules have been provided, hence the support for incremental configuration of the tool. When the tools cannot generate feedback in a certain phase it uses a lower phase to provide an answer.

3.4 Limitations

The decision to only support rule-feedback is made because of time constraints. We believe that a similar framework can also be made for the generation of strategy-feedback, but the research and design of such a framework probably take years.

Phase	Input	Identify mistake by:
1	PT, CT	-
2	0 + Performed rule	The rule the student has supplied
3	0 + Allowed rules	Guessing the rule the student wanted to apply
4	2 + Buggy rules	Path from PT to CT based on allowed- and buggy-rules
PT = previous term, CT = current term.		

Table 3.1: Different phases in rule-feedback generation

Although the tools generated by the framework can directly be used by students, it is not the intended purpose. The generated tools alone are not full-blown educational tools because the command-line interface is not attractive. Furthermore, it is not guaranteed that the principles of Beeson are met because some of the principles mention properties of the solver. Since this solver is not part of the framework we cannot guarantee that it fulfills the properties. Therefore, our current advice is to use the generated tools as a module in an existing solution.

A last limitation is the fact that the tools can only provide rule-feedback given two *well-formed* terms. An example of a term that is not well-formed for the domain of fractions is $\frac{1}{2}+$. This term is not accepted by the parser and does therefore not reach the tools. Providing generic support for the generation of feedback for these kind of mistakes is out of the scope for this thesis.

3.5 Summary

There exists a gap between the research in the area of feedback generation and the current level of feedback in educational tools. Our solution to bridge this gap is to provide a generic framework for the generation of tools, which in turn generate rule-feedback. By implementing the current research in a generic fashion in combination with the possibility of an incremental configuration, we aim at simplifying the generation of tools with high-quality, domain-specific rule-feedback.

Part II

Design

Chapter 4

Rule Feedback Generator

Within the overall design of our generic framework we distinguish two main components: 1) the Rule-Feedback Generator (RFG) and 2) the User Interface (UI). Both of the components are generic in nature, which means that they rely on modules for domain-specific functionality. For example, the rule-feedback generator will need a module that can parse a string to a Abstract Syntax Tree (AST) in order to work.

A overview of the basic interaction between the two components is shown in Figure 4.1. The UI sends the Current Term (CT) and the Previous Term (PT) to the RFG, which returns a feedback message. Notice that this is only the basic interaction; the RFG accepts additional parameters. Under normal circumstances the design of the UI consists of a simple command-line interface. Since all of the inputs can be given as strings we can call the RFG as follows (see Table 4.1 for more information about the available command-line options):

```
$ ./rfg-fraction --previousterm="1/2+1/4" --current-term="2/6"
```

However, this interface is not sufficient to evaluate the RFG with students and teachers. Since they are used to Graphical UI's we need to provide this in order to succeed in real world testing. We have chosen to build our own GUI because there is currently no tool available that supports different domains. When different tools are used for testing the results may vary only because the interfaces are different. This leads to a erroneous image of the usefulness of the RFG's. Therefore, even though it has a low priority, we designed a generic GUI which can be used with different domains. The design of this GUI is discussed in Chapter 5. We discuss the design of the Rule Feedback generator in the rest of this chapter.

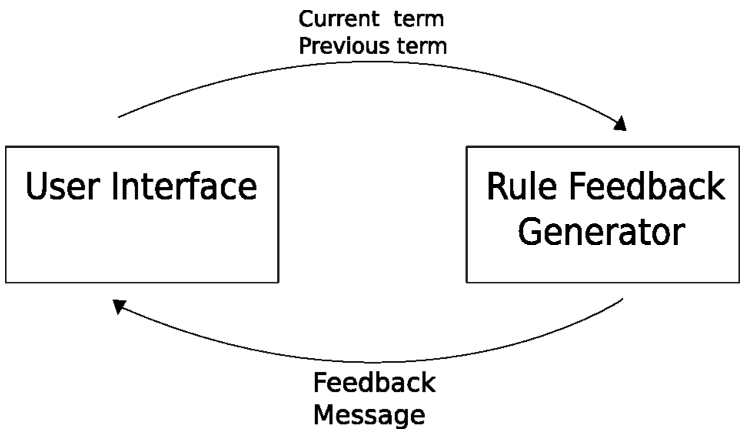


Figure 4.1: Main components of the system

Short option	Long option	Explanation
-H		Shows Help
-V		Shows Version
-C EXPR	–current-term=EXPR	Current Term to process
-P EXPR	–previous-term=EXPR	Previous Term to process
-S RULE	–student-rule=RULE	The rule the student has supplied.
-A FILE	–allowed-rules=FILE	File with allowed rules
-B FILE	–buggy-rules=FILE	File with buggy rules

Table 4.1: Available command line options

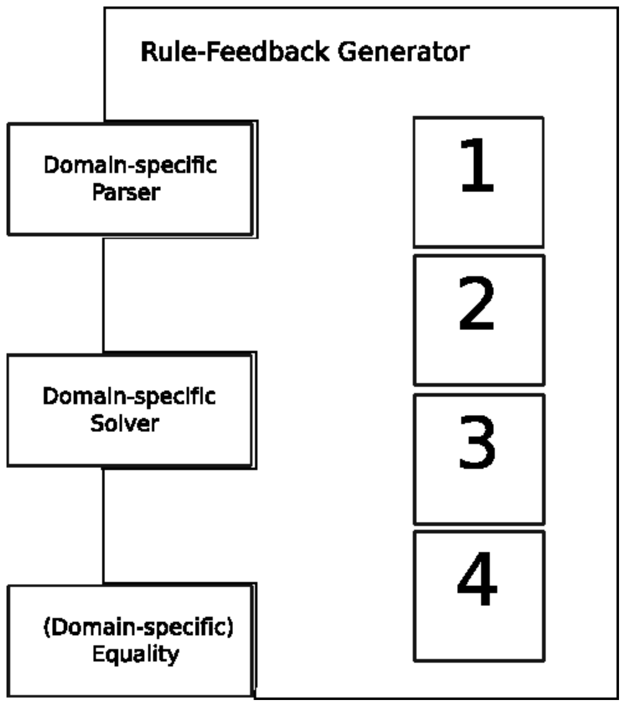


Figure 4.2: Main design of the RFG

4.1 Overall design

The RFG is designed as a generic library that is to be instantiated for different domains. When the RFG is instantiated on for example the domain of fractions we refer to this tool as rfg-*fraction*, or more generally rfg-<*name of domain*>. In this chapter we discuss the design of the different parts within the RFG and how they cooperate.

The overall design of the library is given in Figure 4.2. As suggested by the figure the library contains holes for domain-specific modules. The exact details of these modules are discussed in Section 4.2. Furthermore, the library contains four parts, each representing a single phase of feedback generation. We discuss the design of these phases in Section 4.3 through Section 4.6.

Although Figure 4.2 suggests a linear computation of the feedback, by executing phase 2 after phase 1 is finished and so on, there are several problems with this approach. We will discuss these problems and a possible solution in Section 4.7.

4.2 Instantiation

As mentioned in Section 3.2 we require the following functionality to be available for instantiation. Within the following list, and other examples, we use `RFG a` to represent all of the class restrictions for a given `a`.

```
module RFG-fraction where
import RFG
import RFG-fraction-specific

main = let p = fraction-parser
           e = fraction-equal
           s = fraction-solver
       in main-generate-feedback p s e
```

Figure 4.3: Code for instantiation

- A domain-specific parser that parses a given string to an AST. We refer to this functionality as `<name of domain>-parser` within the example code. The type of this functionality is: `parser :: RFG a => String -> a`
- A domain-specific solver that given an AST representation of an exercise generates the AST that represents the result of the exercise. We refer to this functionality as `<name of domain>-solver` within the example code. The type of this functionality is: `solver :: RFG a => a -> a`
- An optional domain-specific equalizer that answers the question: "are these two AST's equal?". The RFG provides generic equality functionality which is used as default. We refer to this functionality as `<name of domain>-equal` within the example code, even if the generic equal is used. The type of this functionality is: `isEqual :: RFG a => a -> a -> Bool`

Given this functionality, a typical instantiation is described by the code in Figure 4.3, which is essentially a few lines of code to glue the generic functionality to the domain-specific functionality.

4.3 Phase 1

The result of the first phase is a simple *correct/incorrect*-message. This type of message only informs the student whether the previous term and the current term both result in the same answer. The actual content of the message depends on many factors such as language and working environment. Therefore, the content of this message must be configurable.

The input in this phase consists of the PT and the CT. When we assume that both of these terms are already parsed, the algorithm for this phase becomes straight-forward: calculate the answers to both terms and check whether they are equal. If they are equal generate a correct-message, otherwise generate an incorrect-message. Pseudo-code for this algorithm is given in Figure 4.4, where `getConfig s` retrieves the value of the configuration variable `s`.

```
phase1 :: RFG a => Solver a -> Equal a -> a -> a -> String
phase1 s eq ct pt = let res1 = s ct
                        res2 = s pt
                    in if eq res1 res2
                       then getConfig Correct
                       else getConfig Incorrect
```

Figure 4.4: Code for phase 1

```
phase2 :: RFG a => Solver a -> Equal a -> Rule a -> a -> a -> (String,String)
phase2 s eq rl ct pt = let cts         = applyRule rl pt
                           rulemessage = phase1 s eq pt ct
                       in if ct `elem` cts
                          then (rulemessage, getConfig Correct)
                          else (rulemessage, getConfig Incorrect)
```

Figure 4.5: Code for phase 2

4.4 Phase 2

For the second phase an additional input is needed in the form of a rewrite rule. We assume that it is present in a parsed form.

The additional input of the rewrite rule requires the output of two correct/incorrect-messages. The first one judges the application of the rule, the second one judges the rule itself. In order to judge the application we need to apply the rule to the PT and check if any term in the result is exactly equal to the CT. We need to check this on the structure of the AST because we want to know whether the student applied the rule correctly. When we would use the `isEqual` functionality it could be the case that the rule is applied incorrectly, but the term is still semantically equal. This would lead to incorrect feedback about the application of the rule.

For the judging of the rule we can use the algorithm of phase 1 since rules may not change the answer to the exercise. A code implementation for this phase is given in Figure 4.5, where the function `applyRule` is of type `RFG a => Rule a -> a  -> [a]`. This function takes a rule and a term, and returns a list of new terms, one for each of the positions in the AST where the rule can be applied.

4.5 Phase 3

The design of the third phase is based on the assumption that it is only used when it is known that an error has been made. This fact can, for example, be extracted from the result of phase 1. The input for the third phase is the PT, the CT and a non-empty set of

```
phase3 :: RFG a => [Rule a] -> a -> a -> String
phase3 rules ct pt = let rule = case guessRule rules ct pt
                     in  formatMessage (fst rule) ct pt
```

Figure 4.6: Code for phase 3

allowed rules (Γ). The set must be non-empty because guessing an item from an empty set is impossible.

Because the algorithm is used in an educational setting we assume the following properties to hold:

1. The PT and the CT are relatively small (< 100 nodes). Exercises do not use large terms, because large terms confuse the student unnecessarily.

2. Every allowed rule in the set rewrites a term towards the answer-term. So when the rule is applied the resulting term must resemble the desired answer-term more closely than the original term. In other words, each rule in the set is an encoding of an action a teacher would want a student to take.

   We realize that this assumption excludes, for example, rules for commutativity. This is not seen as a problem because a rewrite rule for commutativity is never applied "at random". Such a rule is only applied as part of a larger step which defines the context. For example, it is always allowed to multiply a fraction by any constant. However, multiplying a fraction by a constant is only done when two denominators must be made equal. This context does not only specify when the rule must be applied, it also dictates which constant to use in the multiplication. All of these restrictions can, and must, be captured in a rewrite rule. This corresponds to the fact that a teacher also gives a context while explaining the rule to students. Moreover, the algorithm silently applies rules for associativity and commutativity during rule calculation. This property of the algorithm is discussed more deeply in Section 4.5.1.

3. Because of the small terms, and the notion of associativity and commutativity within the algorithm, we assume the path between the start and the result of the exercise is relatively short (< 15 steps).

4. Given the relatively short path we assume that there are only a few steps taken between the PT and the CT. Moreover, the tool is designed for situations in which teachers advise students to only take a few steps, or even a single step, between two terms.

A helper function for this phase is the function `formatMessage`. This function has a rule, the CT and the PT as input and formats them all into a feedback message. The format of the message is kept abstract, because the composition of the input depends on, for example, language and work-atmosphere. The format of this message is an additional configuration option of the RFG. We can imagine it looks something like this:

> Unfortunately, this step is not correct. I think you wanted to apply this rule: [RULE].
> The left-hand-side of the rule matches [PT], the right-hand-side would become [PTAPPLIEDRULE]. Can you see the difference with [CT]?

Notice that the feedback-message contains holes in the form of `[<holename>]`. These holes are filled with the concrete values of the current exercise and terms. For example, the hole `[PT]` would be filled with the actual PT. More complex, the hole `[PTAPPLIEDRULE]` would be filled with the result of applying the guessed rule, the outcome of the algorithm, to the PT. Since application of a rule can result in several different terms we can take, for example, the term with the lowest distance to the CT.

Apart from the helper function to format the feedback-message this phase also relies on a function called `guessRule` to "guess" the rule a student wanted to apply. This function receives the list of rules, the CT and the PT and returns a rule from the given set. Note that the algorithm can always return a rule because the set is assumed to be non-empty.

Given the aforementioned helper functions the algorithm for the third phase consists of guessing the rule and returning a formatted feedback-message based on this rule. A code representation of this algorithm is given in Figure 4.6. The reason that this algorithm is so concise comes from the fact that the difficult part is hidden in the function `guessRule`. It is easy to see that the quality of the output heavily depends on the quality of the "guessed" rule.

4.5.1 Rule Guessing

In order to produce a feedback message which is useful for a student we need to make a good guess of the intentions of the student. We believe that this is something teachers do all the time when they are correcting errors made by students. We imagine a teacher going through the following process to guess the intentions of a student. Given the fact that the student made an error and the knowledge of the allowed steps, a teacher searches for a step in the calculation of the student that is not correct. When this step is identified the teacher matches this faulty step to an allowed step that looks similar, and gives the student feedback based on this matched step. In case a student writes down something that is not (easily) matched against an allowed action, the teacher asks the student to repeat the calculation in smaller steps.

When analyzing the process of the teacher, we can identify three major steps:

1. Calculate a new rule representing the transformation between the PT and CT, let us call this rule δ.

2. For each rule γ_i in the set of allowed rules Γ, calculate the distance between δ and this rule.

3. Return the rule γ_j which has the lowest distance to δ. When a rule has a close match on the PT but a distant match on the CT, it can share the same distance with a rule which has a distant match on the PT and a close match on the CT. As soon as multiple rules share the same distance, the rule of which the LHS has the smallest distance to the LHS of the rule γ_j is picked [1].

Within the three steps the concepts of "calculating a rewrite rule" and "distance between rewrite rules" are used. We define and formalize these concepts below. With these concepts defined, the three steps above are easily translated into an algorithm. However, such an algorithm would only be able to handle a single rewrite step at a time. When a student takes more than one step the outcome of the algorithm becomes inaccurate.

To counter this inaccuracy we add two ways of supporting multiple rewrite steps: recursion and extension of the rule-set. Using recursion the algorithm can simulate multiple steps on multiple locations. This deals with the case in which a student made a number of steps, any of them faulty. When the student has taken multiple steps which are incorrect, the algorithm returns the rule for the incorrect step which is closest to an allowed rewrite rule. Extending the rule-set supports multiple steps on a single location and can be used to give a more extensive guess. We discuss both of these approaches in more detail later in this section.

Rewrite rule calculation

The calculation of the rewrite-rule between any two terms is done by a bottom-up calculation using the Abstract Syntax Tree (AST) of both terms. Within the root-node of the trees the algorithm first calculates the rewrite rules between the children of the nodes, after which it merges these rewrite rules into a rewrite rule between the current nodes. Because of this recursion there are three different cases to deal with; 1) internal node against internal node, 2) leaf against leaf and 3) internal node against leaf.

Within the first two cases the algorithm checks for equality of the leaves/internal nodes first. If both are equal the algorithm returns the two trees unmodified. Notice that this leads to rewrite rules like the following as a rewrite rule between equal terms:

$$\delta : 1 + 2 + 3 \rightarrow 1 + 2 + 3$$

When two internal nodes are not equal the algorithm needs to match the children of both internal nodes against each other. A straight-forward algorithm is to take the list of children and compute the rewrite rule for each of the pairs from left-to-right. If the number of children differs between the nodes, the additional children are simply kept.

[1] This heuristic is based on the assumption that students are more likely to use rules that can actually be applied.

```
     Plus                Plus                Plus          Var
   /  |  \               /  \                /  \           |
 Num Num Num           Num   Num           Var   Var  ==>   3
  |   |   |             |     |             |     |
  1   4   7             7     3             1     4
```

Figure 4.7: AST representation of $1+4+7$, $7+3$ and the resulting rewrite rule

While the simple algorithm for matching nodes is useful, it does not return the most accurate results. For example, when the PT is $1+4+7$ and the CT is $7+3$ the simple algorithm returns the following as a rewrite rule:

$$\delta : 1+4+7 \rightarrow 7+3$$

However, a better result would be the rule:

$$\delta : 1+4 \rightarrow 3$$

This rule captures the smallest change between the two terms. Since the smallest change reveals the intentions of the student with as little noise as possible, this would be the desired rewrite-rule between the two terms above.

The first step in generating more accurate rules is ignoring equal children. When two internal nodes share a common child the child does not need to be incorporated in the generated rule. The internal nodes themselves can also be ignored when they represent the same operator and have no non-equal children.

A second step in generating the desired result is to adapt the order in which children are matched against each other keeping in mind the properties of the operator-node. When it is known that an operator is associative the children can be matched left-to-right, but also right-to-left. When the amount of children differs for both internal nodes an associative match must also take the smallest list and find the longest common subsequence within the longer list. The match that gives the best result is returned as the result of the computation. In this case the best result is the shortest list of children since equal children are filtered out. Whenever an operator is not only associative but also commutative, all permutations of children are matched against each other. The result is again the shortest list with different children. Notice that this behavior supports the second property mentioned in the beginning of Section 4.5

Integrating the two steps above into the algorithm leads to the desired rewrite rule as a result of the algorithm. In order to show this we give the calculation of the rewrite rule between $1+4+7$ and $7+3$. The original AST representation of these terms is shown in Figure 4.7. The algorithm starts in the root-node of both AST's where it extracts the children of this node. In this case the children for the PT are $\{Num\ 1,\ Num\ 4,\ Num\ 7\}$, and the children for the CT are $\{Num\ 7,\ Num\ 3\}$. When the algorithm examines the properties of the +-operator it discovers that the operator is both associative and commutative. This means that all combinations of children are tried and the best result is kept. In this case the best result boils down to the sets PT' = $\{Num\ 1, Num\ 4\}$ and CT' = $\{Num\ 3\}$. In the case of $PT\ '$ the size of the set is larger then one, therefore the context-node `Plus` is kept. The size of CT' is one and therefore the context-node `Plus` can be ignored in the result. The resulting rewrite rule is also shown in Figure 4.7.

Distance between rules

As shown before, a rewrite rule consists of a term representing the LHS, a term representing the RHS and a set of bindings. By defining the distance between rules as the distance between the LHS of the rules, plus the distance between the RHS of the rules, the distance can be calculated using the distance between terms. As with the calculation of a rewrite rule, the algorithm for term-distance is a bottom-up traversal and needs to define the cases of leaf against leaf, internal node against leaf and internal node against internal node. Furthermore, the algorithm for the term-distance carries an environment, initially holding all restrictions of both rewrite rules. For example, when the distance between the rules:

$$\texttt{SolveMin}: A - B \rightarrow C \quad \texttt{where}\ C := solve(A - B)$$

and:

$$\delta : 1 + 4 \rightarrow 3$$

is calculated, the initial environment is $\{[(C, A - B)]\}$. Note that the *solve*-node is left out in the environment. This is done because solving the restriction on the fly is not always possible. When the RHS of both rules are not a perfect match there might be unbound variables. Since we do want to capture the fact that the variable is bound we have chosen to initially bind the variable to the term in the restriction.

The distance between two leaves depends on the current environment. When either of the leaves is a meta-variable, the environment is checked for a binding. When there is no binding the distance is zero and the meta-variable is bound to the term of the other leaf. In the case of a bound meta-variable, the term to which this variable is bound is taken from the environment and matched against the other leaf. When they are the same the distance is again zero, but there is no need to bind the meta-variable again. If the binding is not equal to the other leaf the distance is $1 + size(term)$, in the case of two leaves this is always 2. No new binding is made in this case. Matching an internal node n against a leaf l is treated in the same way as matching two leaves. Note that when the leaf l is no meta-variable, the distance between the two is $size(n) + size(l)$. Since we already assumed l to be a leaf we can simplify this to $size(n) + 1$.

The last distance to define is the distance between two internal nodes. This distance is defined as *top-level-distance* + *child-distance*. When the internal nodes represent the same operator, i.e. have the same name, the *top-level-distance* is zero, otherwise it is two (one for removing the node, one for adding the other). The *child-distance* is defined as the smallest distance between the sets of children. As with the algorithm for rule-calculation, the order in which children are matched against each other depends on the properties of the operator-node. For each order of matching that is allowed the distance is calculated and the overall smallest distance is taken as the result.

Unfortunately, using a distance of zero for a match against a meta-variable can cause general rewrite rules to be favored. To prevent matches becoming too general, the accuracy of a match is also taken into account. The accuracy is defined as:

$$\frac{total\ size\ of\ bindings}{number\ of\ bindings\ in\ the\ environment}$$

where the size of a binding is the size of the bound term.

To illustrate why the accuracy should be taken into account consider the case in which the calculated rule is:

$$\delta : 1/2 + 1/4 \rightarrow (1+1)/4$$

Furthermore, let the set of allowed rules contain the rules:

`SolveAdd` : $A + B \rightarrow C$ `where` $C := solve(A + B)$
`AddFractions` : $A/B + C/B \rightarrow (A + C)/B$

A match without accuracy would favor the rule `SolveAdd` and return this rule as the one that is closest. This is because the three meta-variables are bound without any costs. In this case, when the matching takes the accuracy into account the closest rule would become `AddFractions`. We believe that a student benefits more from feedback for a more detailed rule, and therefore we choose to take into account the accuracy.

To illustrate the algorithm we calculate the distance between the rule calculated in the previous section:

$$\delta : 1 + 4 \rightarrow 3$$

and the rule for solving subtraction:

`SolveMin` : $A - B \rightarrow C$ `where` $C := solve(A - B)$

The first step in calculating the distance is filling the initial environment. As shown before the initial environment in this case is $\{("C", A - B)\}$. The second step is to take the LHS of δ and match it against the LHS of `SolveMin`. Since both terms consists of a node at the root the names of the nodes are compared. The names are not equal and therefore the *top-level-distance* is two. Since the names are not equal the algorithm can only use the standard left-to-right match of the children. In both cases a free meta-variable is matched against a number, which leads to a distance of zero and a new binding in the environment. The new environment is therefore $\{("C", A - B), ("A", 1), ("B", 4)\}$, which is the environment initially used to match the RHS of both rules. In this case the root-nodes of both terms are leaves, which means that the environment is checked for a binding of the meta-variable $'C'$. Because of the initial restrictions the meta-variable is bound and therefore it cannot be matched against the leaf representing 3. This leads to an extra distance of one for the meta-variable and one for the size of the term that cannot be matched. Examining the last environment gives an accuracy of one, adding this to all the distances leads to a final distance of five.

```
guessRule :: RFG a => [Rule a] -> a -> a -> (Rule,Int)
guessRule rules ct pt =
               let rule'   = calculateRule pt ct
                   guess   = -- calculate current best guess using rule'
                   pts     = concatMap (applyRule pt) rules
                   recDist = map (guessRule rules ct) pts
               in minimumBy distance (guess:recDist)
```

Figure 4.8: Code for general recursion.

Recursion

Now that the algorithm for guessing a single rule has been defined, it can be extended to handle multiple steps. The basic extension is recursion and deals adds support for multiple steps on multiple locations. This allows the algorithm to ignore (a sequence of) correct steps leading towards an incorrect step.

Extending the algorithm with recursion adds a second step to the algorithm. The first step is still to guess the rewrite rule for a single step. After this, new *PTs* are generated by applying a rule from the set of allowed rules to the current *PT*. For each new *PT′*, the rewrite rule to the *CT* is calculated and a guess is made. The final result of the algorithm is the guess with the overall lowest distance. An abstract code implementation of this recursion is given in Figure 4.8.

As an illustration consider the following situation from the domain of logic in which:

$$PT{=}(a \vee b) \Rightarrow c,\ CT{=}(\neg a \wedge b) \vee c$$

Furthermore, the set of rules is:

$$\Gamma{=}\{\ \texttt{MorganOr}: \neg(A \vee B) \rightarrow \neg A \wedge \neg B\ ,\ \texttt{ImpElem}: A \Rightarrow B \rightarrow \neg A \vee B\ \}$$

Using these rules to correctly rewrite the *PT* would result in the term $(\neg a \wedge \neg b) \vee c$. Comparing this term to the *CT* makes us believe that the student applied the rule `ImpElem` correct after which the student forgot the ¬ in front of the *b* during the application of the rule `MorganOr`. The desired result of the algorithm would therefore be the rule `MorganOr`.

The rule between the current *PT* and *CT* is:

$$\delta : (a \vee b) \Rightarrow c \rightarrow (\neg a \wedge b) \vee c$$

Using the algorithm above to calculate the distances between the rules from Γ and the generated rule we get:

$$dist(\delta, MorganO) = 15, dist(\delta, ImpElem) = 9$$

Since the rule ImpElem is a better match this would be the result of the non-extended algorithm.

In contrast, the algorithm with recursion applies each rule that is possible in every position, in this case only the rule ImpElem. When the rule ImpElem is applied to the PT a new PT'=$(\neg(a \vee b) \vee c)$ is built, assuming that the student applied the rule correctly. When the rewrite rule between PT' and CT is calculated the result is

$$\delta_{rec} : \neg(a \vee b) \rightarrow (\neg a \wedge b)$$

Intuitively, this captures the essence of the error made by the student. Examining the set of rules it is clear that the rule MorganO is the rule which was supposed to be applied. The distance calculations confirm this:

$$dist(\delta_{rec}, MorganO) = 4, dist(\delta_{rec}, ImpElem) = 9$$

Since the minimum distance within this recursion is also lower then the minimum distance from the first step, the result of this calculation is the desired rule MorganO.

While recursion makes it possible to handle multiple steps, it introduces a problem. The general problem with recursion is making sure that the algorithm terminates. Fortunately, this problem is solved by the assumption that each rule in the set rewrites the term towards the answer. Because the rules cannot be applied to the term representing the answer, this would violate the property, we know that the rewriting will stop eventually.

Extending the rule-set

Extending the rule-set deals with the situation in which multiple rules are applied on the same location, but one of the rules was applied incorrectly. The extension is done by combing two (or more) existing rules into one combined rule. We can imagine that a student or a teacher prefers feedback that returns a combined rule, instead of either of the single rules which are composed into this combined rule. Feedback based on a combined rule can provide more information, because combined rules contain more information, which benefits the student. To support this kind of feedback the framework allows rule-sets to be automatically extended by combining rules.

The automatic combination of rules is done by matching the RHS of a rule against the LHS of an other rule. When the terms are unifiable, a new rule is made created with the LHS of the first rule and the RHS of the second rule. The restrictions for this rule are the existing restrictions of the two rules plus the new restrictions created by the unification. For example, examine the initial set of rules:

$$\Gamma=\{\ \texttt{Division}: \frac{A1}{B1} / \frac{C1}{D1} \rightarrow \frac{A1}{B1} * \frac{D1}{C1}\ ,\ \texttt{Multiplication}: \frac{A2}{B2} * \frac{C2}{D2} \rightarrow \frac{A2*C2}{B2*D2}\ \}$$

This set can be extended because the RHS of the Division rule is unifiable with the LHS of the Multiplication rule. Therefore, a new rule is added to the set:

$$\texttt{Multiplication.Division}: \frac{A1}{B1} / \frac{C1}{D1} \rightarrow \frac{A2*C2}{B2*D2}\ \texttt{where}\ A1 := A2,$$
$$B1 := B2, D1 := C2, C1 := D2$$

Notice that this merging creates a binding for each variable of the first rule. Furthermore, the names of the rules are combined to ensure uniqueness in the set of rules.

An example of improved output using an extended set of rules is the case in which $PT{=}\frac{3}{5}/\frac{4}{6}$ and the student submitted the $CT{=}\frac{3*4}{30}$. The CT hints that the student applied the rule for division incorrectly, resulting in $\frac{3}{5}*\frac{4}{6}$, after which the rule for multiplication is applied correctly. However, it can also be the other way around.

The calculated rule between these terms is:

$$\delta : \frac{3}{5}/\frac{4}{6} \rightarrow \frac{3*4}{30}$$

When the initial set of rules is used the algorithm guesses the rule `Multiplication`. When the extended set of rules is used the algorithm guesses the new `Multiplication.Division` rule. Whether this merged rule is the desired output depends on the view of the teacher. Our view is that a more accurate rule is preferred, because it gives more information to the student. Since the merged rule has a closer match to both the PT and the CT, otherwise it would not have been chosen, it is more accurate and therefore our desired output.

Even though there are several situations imaginable where this extension is useful, one can also think of scenario's in which it fails. For example, a scenario in which a rule is incorrectly applied in one location, followed by a correct rule in a different location. We validate the usefulness of extending the set of rules in our evaluation.

Limitations

The performance of the algorithm depends heavily on the algorithm for calculating the distance between terms. Unfortunately, the current algorithm is not capable of coping with large precedence changes. Forgetting parentheses around a sub-term is only a small mistake in a textual view, the consequences for the AST can be severe. A small change in precedence has the ability to change the AST dramatically, resulting in a large distance between two terms and therefore between two rules. Whether this flaw influences the algorithm too much to be useful is considered in the evaluation section.

Furthermore, since the algorithm always returns a rule from the given set, the returned rule is not always linked with the terms provided to the algorithm. It can simply be the case that there is only a single rule in the set, this rule will than always be chosen as the result. This limitation is also considered in the evaluation section.

Summary

The algorithms for calculating a rewrite rule and calculating the distance between terms are relatively straightforward. This results in a straightforward algorithm for guessing a single step. In order to support multiple steps the algorithm is extended by recursion and the extension of the set of rules. The overall result is an algorithm which does not need a great deal of information to return feedback.

```
phase4 :: RFG a => [Rule] -> [Rule] -> a -> a -> Maybe String
phase4 rules buggyrules ct pt =
                    case guessBuggyRule buggyrules rules ct pt of
                           Nothing    -> Nothing
                           Just rule  -> Just (formatMessage rule ct pt)
```

Figure 4.9: Code for phase 4

4.6 Phase 4

The algorithm of the fourth phase is based on the work of Hennecke [18], and adapted to fit into the generic approach of the RFG. As with phase 3 we assume that this phase is only used when an error has been made. Furthermore, we (again) assume the existence of a set of rules which encode allowed behavior. In addition we assume the existence of a set of rules which encode known mistakes, so called *buggy-rules*. In addition to the description of the rule, a buggy-rule can optionally hold a custom feedback-message. One can imagine that such a feedback-message can be entirely adapted to a specific class-room, or even to a single student.

Pseudo-code for the algorithm of this phase is shown in Figure 4.9. This code closely resembles the code in Figure 4.6; the only difference is the additional parameter to the main function and the `guessBuggyRule` function. Notice that the `formatMessage` function still receives only a single rule, but it can now also access the optional feedback-message when this rule is a buggy-rule. The semantics of this function still remains the same.

As is to be expected, the quality of this phase also depends on the quality of the rule returned by the function `guessBuggyRule`. This function receives a set of buggy-rules as additional parameter, we refer to this set as Ω. Recall that the set of allowed rules is referred to as Γ.

Within phase 4 the task is to identify a rule ω which most accurately describes the error made by the student. When we assume that there is a rule in Ω that precisely matches the error the algorithm is simple. We calculate the application of every $\omega \in \Omega$ to the PT and compare the result with the CT. The result of the algorithm becomes the ω which results in an exact match.

Even though an exact match is possible, we have to deal with the case in which there is no single buggy-rule which explains the error. Let us assume that the student made a series of steps, both correct and incorrect ones. This means that

$$\exists \phi_1, \ldots, \phi_n \in (\Gamma \cup \Omega)(\phi_n \, . \, \ldots \, . \, \phi_1\,)PT = CT.$$

The path above can be found by recursively applying the rules from both sets to the PT until the CT has been found. By keeping track of the applied rules in two sets, one for allowed rules and one for buggy-rules, we eliminate paths that only differ in their ordering. When the CT has been found we inspect the set of buggy-rules that have been applied. If this is a singleton set we arrive at our first case and we return the rule within the set. Otherwise, we have to choose a rule from the set of buggy-rules.

The decision for a particular buggy-rule is made by looking at the values of the *assessment* of the rules. The initial value of these assessments are defined in the definition of the buggy-rules, and represent the chance of a student applying this rule. When the value of the highest assessment is higher then the second-best value plus a certain K, a number which is an input to the RFG, then the rule belonging to the highest assessment is chosen from the list. Otherwise, a factor of $\frac{1}{\text{<number of found rules>}}$ is added to the assessment of all the rules in the current list, without returning a result. For example, if there are three rules with an initial values:

$$assess(1) = 0.4, assess(2) = assess(3) = 0.3$$

and $K = 0.2$. Within a first exercise the algorithm finds the set $\{1, 2\}$ and updates the assessments of these rules to:

$$assess(1) = 0.6, assess(2) = 0.45$$

In the second exercise the algorithm finds $\{1, 3\}$ and sees that:

$$assess(1) > (assess(3) + K)$$

so rule 1 is chosen as the result of the algorithm.

To illustrate the complete algorithm, let us look at an example in which:

$$PT{=}\frac{1}{2} + \frac{1}{3}, CT{=}\frac{2}{5} \text{ and } K = 0.2$$

Furthermore, the set of allowed rules is:

$$\begin{aligned}\Gamma{=}\{\ & \texttt{AddDiffFractions}: \frac{A}{B} + \frac{C}{D} \rightarrow \frac{A*D+C*B}{B*D}\ ,\\ & \texttt{SolveAdd}: A + B \rightarrow C \quad \texttt{where}\ C = solve(A + B)\\ & \texttt{SolveMul}: A * B \rightarrow C \quad \texttt{where}\ C = solve(A * B)\ \ \}\end{aligned}$$

Lastly, we use the following buggy-rule:

`AddError, You can only add fractions with equal denominators (0.3)`
$: \frac{A}{B} + \frac{C}{D} \rightarrow \frac{A+C}{B+D}$

This is also the rule that we want to get as result. When we iterate over all the rules from both sets the result is the following set:

$$\{(\tfrac{5}{6}, \texttt{[AddDiffFractions,SolveMul,SolveMul,SolveMul,SolveAdd]}, \texttt{[]})$$
$$, (\tfrac{2}{5}, \texttt{[SolveAdd,SolveAdd]}, \texttt{[AddError]})\}.$$

Only the last term is equal to the CT and has a single buggy-rule in the sequence of application. This makes the rule `AddError` the result of the algorithm.

To illustrate the updating of the assessments we add the following rule to our set of buggy-rules:

`TopAddError, Remember that the denominators need to be equal (0.2)`
: $\frac{A}{B} + \frac{A}{D} \rightarrow \frac{2*A}{B+D}$

Using this extra buggy-rule the result-set after iteration would be extended with:

$\{(\frac{5}{6}$, `[AddDiffFractions,SolveMul,SolveMul,SolveMul,SolveAdd]`, `[]`$)$
, $(\frac{2}{5}$, `[SolveAdd,SolveAdd]`, `[AddError]`$)$
, $(\frac{2}{5}$, `[SolveMul,SolveAdd]`, `[TopAddError]`$)\}$.

In this case there are two distinct buggy-rules found. When we examine the assessment of these rules we see that:

$$|assess(AddError) - assess(TopAddError)| = 0.1 < K$$

Therefore, we do not return a rule as a result. The only thing that is done is updating the assessment of each of the rules by a factor of $\frac{1}{2}$:

$$assess(AddError) = 0.3 * 1\tfrac{1}{2} = 0.45,$$
$$assess(TopAddError) = 0.2 * 1\tfrac{1}{2} = 0.3$$

4.7 Phase Collaboration

From the algorithms of phase 1 and phase 2 we can deduct that these phases always terminate within a certain time-frame, they only consist of one or two non-recursive steps. However, this property cannot be guaranteed for phase 3 and phase 4. Within these phases we know that the algorithm eventually stops, either because there is an answer or there is no rule that can be applied, but it can take a considerable amount of time. Since we aim at providing immediate feedback we need to limit the running time of these phases. When a limit is reached we either take the current best guess, or fall back to the output of an earlier phase.

Limiting the running time can be established by placing an upper-bound on, for example, the running wall-clock time, the number of steps to take in the phases or the number of generated terms to consider in the algorithm(s). The first restriction limits the running time in a domain-independent way, while the second and third restrictions are more domain-specific. These restrictions make sure that a fixed number of terms are considered, but a domain with large terms can have a considerably longer running time because it takes more time to compute, for example, the rewrite rule between two terms.

With the restrictions on certain resources we need to define an evaluation order for the phases. A naive order of executing the highest phase first, then the second highest phase, and so on can result in sub-optimal feedback or no feedback at all. It could for example be the case that there does not exist a buggy-rule which can rewrite the PT to the CT. In this case the fourth phase may consume all resources, leaving the program without any output to give.

Keeping in mind the fact that we at least give a correct-/incorrect-message, and the fact that information from the student is valuable, we define the following evaluation order. When we receive a rule from the student we evaluate the second phase and the second phase only. It is no use to guess a rule the student wanted to apply if we already know what the student wanted to do. In the case that the student does not provide a rule we first execute the first phase. If the result is a correct-message we are done, otherwise we start the third and fourth phase. Note that we can only start them when there are rules for these phases. Because both the third and the fourth phase generate sets of terms by applying rules, we can imagine that parts of these sets can be shared amongst the algorithms. As soon as a resource reaches its limit we return the result of the highest phase available.

An additional way to limit the running time of the third phase is to break the recursion when the best guess of the current iteration has a larger distance than the best guess of the previous iteration. For example, in iteration N the best guess has a distance of 5. This iteration goes into recursion by calling the *guessRule* function for every new PT with this distance. The iteration $N + 1$ first calculates the best guess based on the given PT, in this example the distance to the best guess is 6. Since this distance is greater than the distance of the iteration N we break the recursion and immediately return the best guess. Unfortunately, this breaking of the recursion restricts the algorithm to only finding a local minimum. However, using the assumption that a student only takes a few steps before submitting a new CT, and the assumption that an exercise is generally solved in a small number of steps, we believe that the local minimum is generally also the global minimum.

4.8 Summary

The design of the RFG mostly consists of the algorithms for the different phases of feedback generation. These phases use domain-specific functions given to the RFG on instantiation. The first two phases use simple, non-recursive algorithms to generate feedback on the level of correct/incorrect. Phases three and four are capable of returning more fine-grained feedback, only requiring sets of rules as input. The algorithms do not assume that the sets contain all rules of a domain, they only assume that the sets contain at least a single rule.

In order to limit the running time of the RFG, restrictions can be placed on a number of resources. By making these restrictions configurable the tool can be adapted to fit a single class-room or even a single student.

Chapter 5

Graphical User Interface

In this chapter we discuss the design of the other component of the RFG, the Graphical User Interface (GUI). As mentioned in Chapter 4 we do not need a GUI for the correct functioning of the RFG. To be more precise, the GUI is completely separated from the RFG, it only exists for testing purposes. The RFG itself can be plugged into an existing solution, as is our intention.

Since the interface is only constructed for testing purposes it does not need to be the "ideal" GUI. However, we cannot expect students and teachers to use a minimal interface, else we could have used the command-line interface. In order to stay close to current interfaces of tools we have taken our observations from Section 2.3 into account. Furthermore, guidelines and suggestions from the literature [14, 16, 19, 20, 32, 35] are taken into account. Note that our observations from Section 2.3 may not be in-line with philosophies from the literature, in this case we let the literature take precedence.

The GUI described here is designed to be generic so that it can be used for different domains. This means that the basic components look the same for all domains, only a few are domain-specific. By providing an uniform access to the RFG over all domains, we hope to minimize the influence of the interface on the evaluation of the RFG.

5.1 Screens

We introduce the GUI by means of a user-scenario involving a fictional student. This student has successfully logged in and has selected a domain to practice within, in this case the domain of fractions. After this selection the student is redirected to the initial screen shown in Figure 5.1. This figure shows several important areas, denoted by red circles, which we explain below.

1. The student is greeted by name to give the application a personal touch.

2. In order to keep track of the overall progress the screen shows how much work there is left to do. According to Chalmers [10] this helps the student to keep track of his work.

3. Every student can choose between three different representations of the exercise. The first one is a *graphical* representation which is similar to the notation used on paper. A second one is the more abstract *text* representation which is used as input to the RFG. The last view is a combination of both representations in a *merged* view. This enables more experienced users to work quickly, while it also gives novice users a chance to use familiar notation. Providing multiple ways to input data is recommended by Schneiderman [34]. By keeping the views consistent, a change in one representation is reflected in the other, the student can gradually move from the graphical to the textual interface.

4. The representation of the current exercise is done by a large font with a distinctive color from the background. Using distinctive colors to indicate the current term and soft colors for the background is mentioned by multiple sources.

5. By placing the cursor (and therefore the focus) on the area in which the answer is to be entered, we give the student a visual clue on where to start [16]. Notice that the student enters the answer *below* the previous step. This makes sure that the student can read the work from top-to-bottom, the direction in which most students read, and mimics the pencil-and-paper approach.

6. In order to support the graphical representation the student has access to a virtual keyboard which can be used to enter symbols which are not on the keyboard. Of course, the representation of symbols is domain-specific, but the looks of the keyboard remains consistent over different domains. The initial placement of this keyboard is in the upper-right corner, but the student can place it anywhere on the screen.

7. The arrow does not only indicate the direction in which the student works, it initially represents the equality between terms. This means that the arrow can be read as: *the upper term is equal to the bottom term*.

The looks of the GUI after selecting the merge-view are shown in Figure 5.2. The student now sees the text representation of the term at (1). When an answer is entered in text-format at (2), the visual representation of the answer is also updated. This gives students the chance to experiment with both representations of the same term and understand their relation.

The design in Figure 5.2 is an example of hiding certain parts of the domain until the student is capable of understanding their meaning [16]. When a student does not understand the textual representation it can simply be hidden by choosing the "graphical" tab. Another example of hiding parts of the domain is shown in Figure 5.3, where our student has clicked on the arrow at (1). The arrow now expands to the right and allows the student to enter a rule that represents the action the student wants to perform. How this rule is entered, in a text or a graphical representation, depends on the current view and the domain. Notice that this rule is needed in the second phase of the RFG.

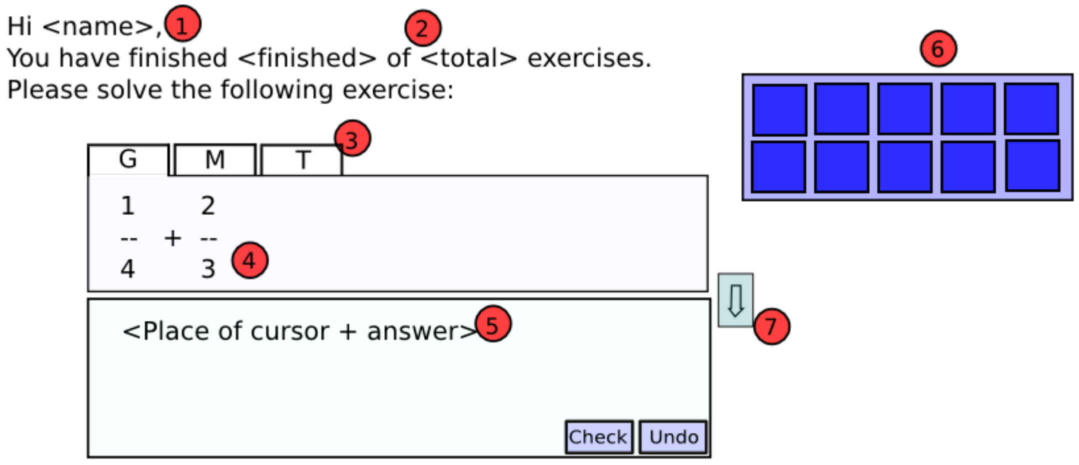

Hi <name>,
You have finished <finished> of <total> exercises.
Please solve the following exercise:
G
M
T
<Place of cursor + answer>
Check
Undo
1
2
3
4
5
6
7

Figure 5.1: Initial screen

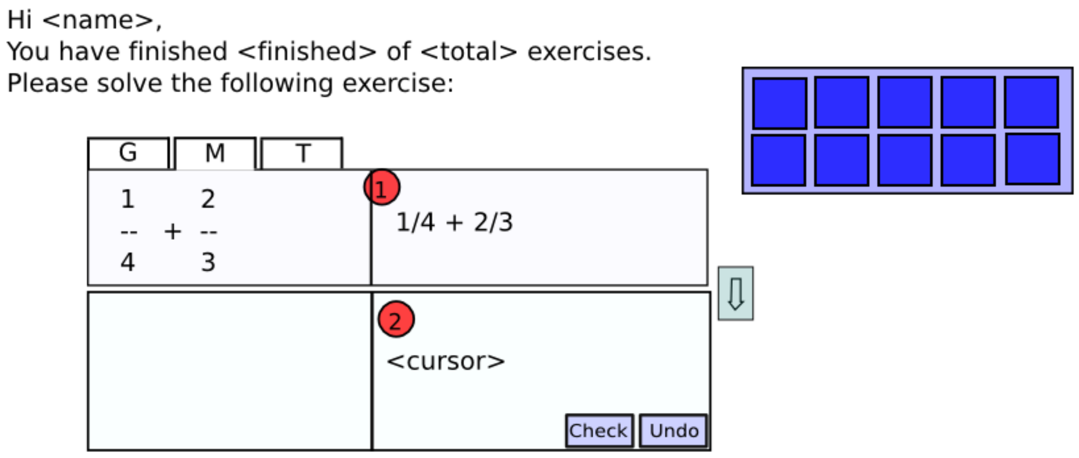

Hi <name>,
You have finished <finished> of <total> exercises.
Please solve the following exercise:
G
M
T
1/4 + 2/3
<cursor>
Check
Undo
1
2

Figure 5.2: Screen with merged view

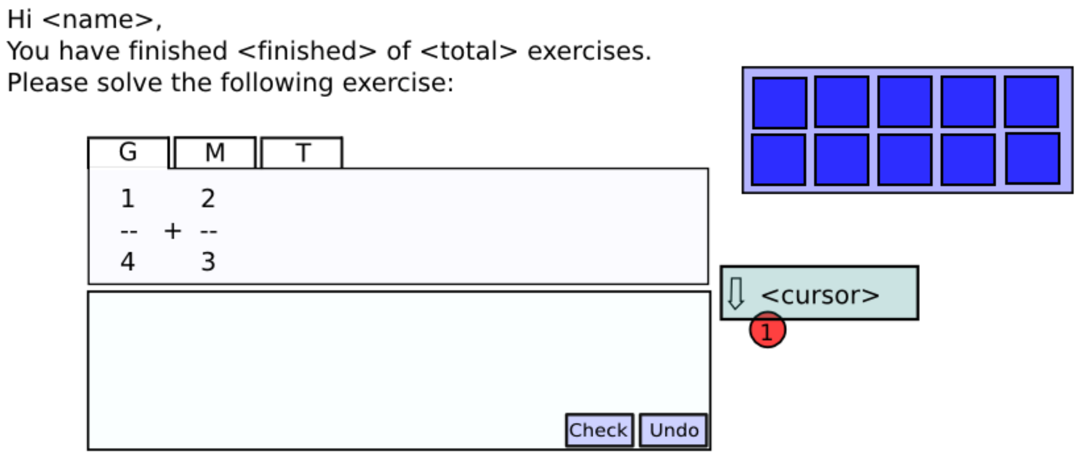

Hi <name>,
You have finished <finished> of <total> exercises.
Please solve the following exercise:
G
M
T
1 2
-- + --
4 3
<cursor>
1
Check
Undo

Figure 5.3: Screen with rule

In our current scenario the student does not want to enter a rule and clicks back to the answer area. This gives the arrow its old semantics of … *is equal to* … back. Our student (finally) begins with the exercise and rewrites the exercise into the term shown in Figure 5.4 at (1). Note that the new term is not a correct rewrite, a fact which influences the behavior at (2). Normally, this area allows another rewrite to be entered when the exercise is not finished yet. In the case of an error this area can be disabled by the teacher to disallow further calculations until the error is resolved. This is one example of adapting the application to a class-room setting, something which is greatly appreciated by teachers [22]. Finally, we see the area in which the feedback is displayed at (3). The feedback is located visually close to the term and in between the terms. By giving the feedback at this location and with a different background-color we minimize the chance a student overlooks the feedback.

From the perspective of the RFG the most important item of the interface is showing the feedback. The feedback is generated from the input entered in the answer area and optionally in the area besides the arrow. Note that when the student operates in graphical mode the input to the RFG is still a text-string, the graphical view is really just a different representation. The interface is designed to send the input to the RFG as soon as the *check*-button is clicked. The *undo*-button allows the student to go back to any previous state.

Our current recommendation is to use the RFG as a module within existing tools. Therefore, the design of the interface only focuses on the ability to show feedback. Interfaces for other components of educational tools, for example lists of students and classes, average scores of students and manipulation of configuration options, are ignored. These components and their interface are out of the scope for this thesis.

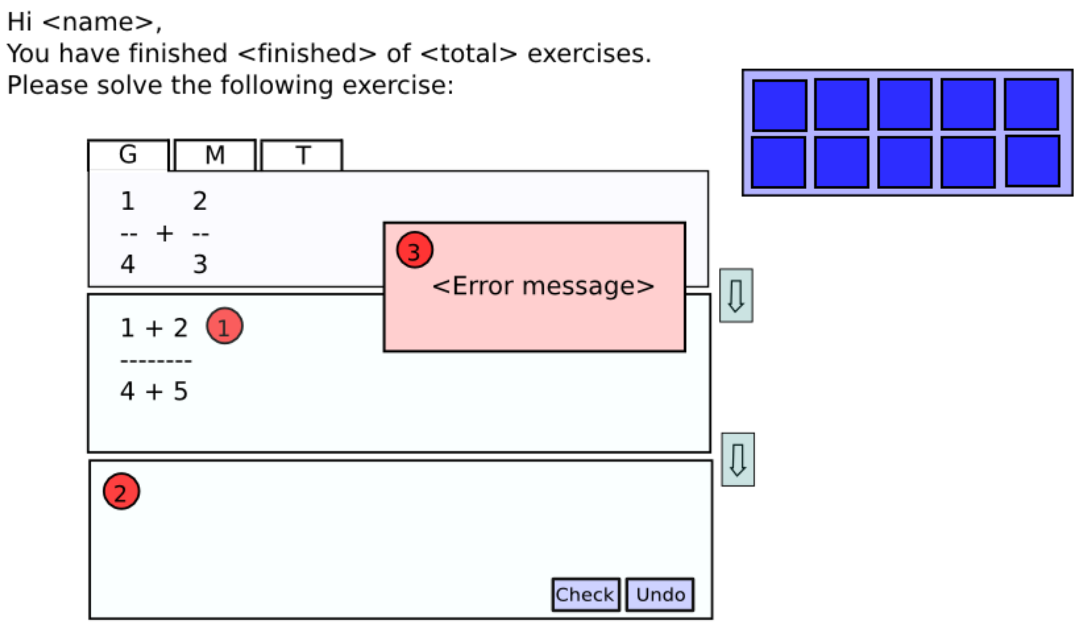

Hi <name>,
You have finished <finished> of <total> exercises.
Please solve the following exercise:
G
M
T
1 2
-- + --
4 3
1 + 2
----
4 + 5
<Error message>
1
2
3
Check
Undo

Figure 5.4: Screen with error screen

5.2 Summary

We have discussed the design of a generic interface for the RFG. This interface is used for testing purposes and therefore not meant to be perfect. However, the design still adheres to many of the recommendations found in current literature. Even though this design only targets a small aspect of the interface for an educational tool, the RFG can be tested with only these aspects.

Part III

Implementation

Chapter 6

Rule Feedback Generator

The framework is implemented in the functional programming language Haskell[13]. This typed languages allows us to generalize over data-types which makes it easier to implement generic functionality. Furthermore, there are modules available for pretty-printing, parsing and testing. Using these libraries speeds up development and makes it easier to verify the code. Moreover, the libraries make it easier to add new domains.

Even though there are some modules available for rewriting, none of these modules can be used in this setting because they all rely on functions defined in Haskell itself to rewrite their input. Ideally, the sets of rules that are used are specific for a single student, a situation which makes it unpractical to define rules within source code. Therefore, we have developed a module for parsing, applying and calculating rules, based on the assumption that the rules are applied to instances of the type class *RFG*. The syntax and semantics of our implementation of rules can be found in Appendix A, the type class and its use is discussed in Section 6.1.

Using the type class, and the available libraries, a new domain can be added without much effort. To show this, we show the steps needed to instantiate a new domain in Section 6.2. The implementation of the different phases is a straightforward translation of the design interleaved with code for bookkeeping. Therefore we do not discuss the implementation of the phases. Interested readers can download the code from:

`http://svn.cs.uu.nl:12443/repos/rfg/`

Within this implementation there is an additional initialization phase. We introduce the functionality and the behavior of this phase in Section 6.3. Finally, Section 6.4 discusses the overall structure of the implementation.

6.1 Type class for rewriting

The definition of the type class needed for the implementation of the RFG is given in Figure 6.1. This definition shows that the super-classes of any RFG data-type are `Show`, `Eq` and `Ord`. The data-type needs to be shown in the feedback message and equivalence on the structure is needed in the second phase. The restriction that the type needs to be

```
class (Show a, Eq a, Ord a) => RFG a where
    isMetaVar   :: a -> Maybe String

    makeMetaVar :: String -> a

    isAssociative :: a -> Bool
    isAssociative _ = False

    isCommutative :: a -> Bool
    isCommutative _ = False

    crush :: a -> (String,[a], [a] -> a)
```

Figure 6.1: RFG type class

an instance of Ord is because of the checks within rule(-merging).

Defining generic rewrite rules is usually done by using meta-variables. To be type-correct a meta-variable needs to be part of the data-type over which a generic rule is defined. Since it is not possible to extend an already defined data-type with a new constructor, and we do not want to force a certain constructor to be available, the type class contains two functions to abstract over meta-variables. The first function needs to inspect a value to see whether it is a meta-variable, the other must create a meta-variable given a name. This way, an implementer can choose to use a separate constructor, or to use an existing constructor with magic values as a meta-variable.

Two functions that inspect the data-type need to be defined for the third phase. The functions isAssociative and isCommutative take a value and determine whether or not the top-level constructor encodes an operator that is either associative or communicative. The results of these functions are used in the function to calculate a rewrite rule. As explained in the design of this algorithm, knowing that an operator has certain properties makes it possible to calculate rules that encode the smallest change. These functions have a default implementation, which always returns False, to make the functions optional.

The last function in the type class is the function to break down a constructor for inspection. The function crush takes a term and should return a triple encoding the name of the constructor, the list of children of this constructor and a function to construct a term with the constructor giving a list of children. An element that might seem strange in this triple is the name of the constructor. This value is needed in order to calculate whether two top-level constructors are equal, a fact needed by, for example, the algorithm for calculating the distance between nodes. The other elements of the triple, the children and the build-function, are used extensively throughout the rules module. For example, the children of a constructor are used to find all places in which a rule can be applied by recursively applying the function applyRule over the children of a term.

Furthermore, having access to the children makes it possible to calculate the size of a term in only a few lines of code:

```
size :: RFG a => a -> Int
size x = let (_,chlds,_) = crush x
         in 1 + (sum.map size) chlds
```

The last two elements in the triple returned by the function `crush` are similar to the functionality described by Mitchell [27] in his paper about Uniplate. Although some of the functionality of Uniplate could have been used in our implementation, Uniplate does not support the comparison of constructors without taking the children of the constructor into account.

6.2 Instantiating a domain

To show the steps for instantiating a domain we show the steps made to instantiate the domain of logic using the framework. The first step in making the domain available is the creation of the data-type for logical expressions. The data-type that we use contains the usual constructors and is shown in Figure 6.2. Notice that the `Or`-constructor and the `And`-constructor have a list of children instead of only two children. Limiting these constructors to having only two children leads to a nesting of these constructors in, for example, the term $p \wedge q \wedge r$. This would lead to a distance between the previous term and the term $(p \wedge q) \wedge r$ only because of the nesting. Since both terms are exactly the same we have chosen to encode this property explicitly. Furthermore, we have chosen to add a special constructor to encode meta-variables. An alternative would be to use the `Literal`-constructor with upper-case characters as meta-variables. This saves one constructor, but the implementation becomes less clear. The instance of the RFG type class that belongs to this data-type is shown in Figure 6.3. Since this is a straightforward translation from the data-type declaration we can imagine that in the future the instances for RFG can be derived from the data-type itself.

When the data-type is declared, the following step is to create a parser to parse a string to this data-type. Actually, the parser that needs to be supplied should be an extensible parser, the function should have the type:

```
logicParser ::  RFGParser a -> RFGParser a
```

The reason for forcing the parser to take a parser as an argument is to make sure that the same parser can be used for both terms and rules. Parsing terms is done by supplying a parser that always fails, parsing rules is done by supplying a parser that parses meta-variables. Within the implementation of the function the supplied parser should be used in places where a meta-variable has a semantic meaning. For example, within the domain of logic it only makes sense to parse a meta-variable on the same level as a literal. Therefore, the supplied parser is used only as an alternative for when the parsing of a literal fails.

```
data Expr = Literal Char
          | TRUE
          | FALSE
          | And     [Expr]
          | Or      [Expr]
          | Not     Expr
          | Impl    Expr Expr
          | Equal   Expr Expr
          | MetaVar String deriving (Eq, Ord, Show)
```

Figure 6.2: Logic data-type

```
instance RFG Expr where
  makeMetaVar str = MetaVar str

  isMetaVar (MetaVar str) = Just str
  isMetaVar _             = Nothing

  crush (Literal x)     = ("Literal" ++ [x] , []    , const (Literal x)      )
  crush (MetaVar x)     = ("MetaVar" ++ x   , []    , const (MetaVar x)      )
  crush (TRUE)          = ("TRUE"           , []    , const TRUE             )
  crush (FALSE)         = ("FALSE"          , []    , const FALSE            )
  crush (Not     x)     = ("Not"            , [x]   , \[x]      -> Not     x  )
  crush (And     c)     = ("And"            , c     , \x        -> And     x  )
  crush (Or      c)     = ("Or"             , c     , \x        -> Or      x  )
  crush (Impl    x y)   = ("Impl"           , [x,y] , \(x:y:[]) -> Impl    x y)
  crush (Equal   x y)   = ("Equal"          , [x,y] , \(x:y:[]) -> Equal   x y)

  isCommutative (Or  _) = True
  isCommutative (And _) = True
  isCommutative _       = False

  isAssociative (Or  _ ) = True
  isAssociative (And _ ) = True
  isAssociative _        = False
```

Figure 6.3: RFG Expr instance

Within the current implementation, the parsing is done by using the ParSec [21] parser-combinators. These combinators are used not only because they are part of the standard Haskell-libraries, they also come with standard support for parsing expressions. In order to create the parser for logic we only needed to provide the operators, their symbols and the precedence between them.

The third step in the process is to create a solver for our data-type. In our case we define the solver to rewrite any `Expr` into a new `Expr` representing the Disjunctive Normal Form (DNF) of the term. This is done in a module that takes a term, matches on the top-level constructor and calls the appropriate rewrite function on it. We have chosen this approach instead of exhaustively applying allowed rewrite rules to the term because of speed-issues. Implementing the rules directly in Haskell saves the overhead of matching rules to terms.

As a final step, the equality function between terms needs to be defined. As suggested by the data-type we simply use the standard equality function for this by deriving the `Eq` class for our data-type.

This concludes the instantiation of the domain. By instantiating the data-type on the RFG-class the functionality of calculating rules, calculating term-distance and applying rules becomes available. Furthermore, the extensible parser allows the framework to parse both rules and terms alike. Lastly, the solver and the (derived) equality functionality make the data-type available to all the four phases of the framework.

6.3 Phase 0

This bootstrapping phase handles the administration of the tool before starting the feedback generation. It checks whether the required PT and CT are available after which it calls the given parser on these terms. When file-names are given for the sets of allowed and/or buggy-rules these files are parsed as well. Any error in parsing stops the executing of the generated tool with a parse error.

To avoid unnecessary work, this phase also checks whether the given CT is the answer to the exercise. This is done by first solving the CT and the PT with the given solver, resulting in a CT' and PT'. The resulting terms are compared against each other to make sure that the results are the same and thus no error has been made. After this, the supplied equality function is used to check whether the CT and the solved CT' are semantically equal. When this is the case a *Done*-message is generated.

6.4 General structure

The implementation of the framework together with the different instantiations for the three domains is separated into four parts. As is to be expected, three of these parts are domain-specific and one contains the common functionality of the framework.

Each of the domain-specific parts is made up out of four modules, a data-module, a solver-module, a parser-module and a main-module. The functionality of each of these

Name	Contains
Config.hs	Configuration options, i.e. feedback-messages
FirstPhase.hs	Handling of the first phase
Fourthphase.hs	Handling of the fourth phase
Main.hs	Calling different phases based on input, phase 0
Options.hs	Declaration of options that can be given to the tools
Parser.hs	General parser functionality, i.e. parser for rules
Prelude.hs	Small helper functions
Rules.hs	Applying, calculating and calculating distance between rules
SecondPhase.hs	Handling of the second phase
ThirdPhase.hs	Handling of the third phase
Types.hs	Type-synonyms, class declarations
Utils.hs	Helper functions for the instances of the RFG-class

Table 6.1: List of modules of the framework

modules is derivable from the names. In all three cases the equality is the standard `Eq` behavior. Since this equality is described by the operator `(==)` there is no separate module for this.

The common part of the framework consists of twelve modules, their names and purposes are listed in Table 6.1. Note that the module `Config.hs` currently contains all of the configuration of the framework. We can imagine that this information will in the future be stored outside of the actual code to make the tool easier to adapt for a single classroom.

Within the current implementation the code in the module `Main.hs` calls the different phases based on which information is available. For example, when a rule is supplied the code only instantiates the second phase.

Currently, the main function leaves some room for optimization. When the tool is called with parameters for both allowed rules as well as buggy rules it first runs the fourth phase, only initiating the third phase when the fourth phase does not return an answer. To somewhat limit the fourth phase, we have only implemented a parameter representing the number of recursive steps to be taken. We can imagine that the functionality for both the third phase and the fourth phase are merged into a single function.

6.5 Summary

We have explained the structure of the type class that is needed for using the framework. We have given an example of a data-type with its class instantiation for the domain of logic. Furthermore, we discussed the steps needed to instantiate the framework for this domain. An initialization phase is discussed followed by an overview of the structure of the implementation. Although the current implementation leaves some room for optimization, it is useful for testing purposes and as a reference implementation.

Chapter 7

Graphical User Interface

The implementation of the GUI is drastically influenced by a single technical decision, the decision whether the application is to be run stand-alone or as a web-application. When we base this decision purely on the guidelines from Section 2.3 the application would run stand-alone because that makes it easier to respect these guidelines. For example, techniques for precisely positioning elements are available in a number of libraries. Furthermore, a stand-alone application generally gives a more responsive look-and-feel, simply because there are more resources available on the machine. On the other hand, there are also techniques available to make a feel web-application more responsive. Still, positioning elements is harder in a web-application because of different browsers, but it is not impossible.

Although a web-application makes it harder to respect the guidelines from Section 2.3 we are still in favor of this approach. The last report on ICT in education [26] of the Dutch government confirms the belief that this is the right choice. It states that educational software must be "... available and accessible through public and on-line channels". Providing a web-interface also keeps the code in a single place. This makes it easier to update the functionality and fix bugs. Furthermore, the problems associated with distribution and installation of the tools are nullified. A last positive point is that a single point of entrance makes it easier to log the activities of students and teachers, something which is useful for evaluation purposes and analysis.

The choice for making a web-application splits the GUI into two components, the client-side and the server-side. The client-side is the component in which the user enters the input which is sent to the server-side. At the server-side, a script processes the user-input and calls the RFG with the appropriate parameters. The message that is returned from the RFG is inspected by the script and the eventual feedback is send back to the client-side component which displays the message. This information flow between these two components is shown in Figure 7.1.

Within the rest of this section we discuss some of the techniques used to implement the GUI. In Section 7.1 we discuss the client-side component, the actual interface the testers see. The techniques of the server-side component are discussed in Section 7.2. Section 7.3 concludes.

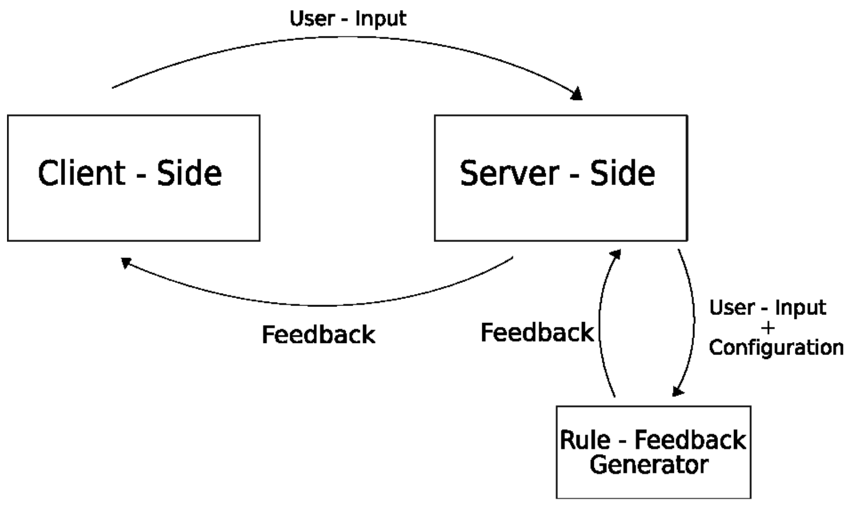


Figure 7.1: Components of the GUI

7.1 Client Side

Because of the nature of a web-application the graphical parts of the client-side components are defined in HyperText Markup Language (HTML) and styled using Cascading Style Sheets (CSS). The HTML-components are generated on the server-side by a script written in PHP[1], a widely-used, general-purpose scripting language. We have chosen to generate the HTML-components with absolute positions to keep the same look-and-feel across different browsers. Using absolute positions made sure that the interface greatly resembles the figures in Chapter 5.

The support of the textual representation is already defined by the HTML-standard. Unfortunately, a graphical representation of, for example, mathematics is not supported by default. Furthermore, dynamically updating a web-page to keep both representations synchronized can not be done within HTML itself. Fortunately, this can be done by using the client-side scripting language JavaScript. Most browsers support this scripting language, although the implementations vary across browsers. Fortunately, the JavaScript framework Prototype[2] enables us to write scripts without having to worry about these differences. The problem of representing mathematics visually is solved by using the jsMath[3] package. This package is capable of transforming TeX, a widely used typesetting system, into HTML on the fly. After creating a small stack-based parser in JavaScript we were able to take the textual representation of a term, transform it into TeX-code, generate HTML-code and update the visual view all on the client-side. This

[1]http://www.php.net/
[2]http://www.prototypejs.org/
[3]http://www.math.union.edu/ dpvc/jsMath/

not only saves resources on the server, but it also makes the application feel more responsive.

Since the representations of the terms differ across domains the client-side component needs a domain-specific keyboard and parser. These two components need to be provided to the script that generates the HTML-code in order to instantiate the GUI for a domain. This process is similar to instantiating the RFG-framework.

When the student has filled in the new current term, and possibly the rule used to get this term, a button can be clicked to receive feedback. The button is a normal HTML-button that submits the form containing the input from the user to the server-side component.

7.2 Server Side

On the server-side we use PHP to process the user input. This language is chosen because it allows for rapid prototyping without much configuration issues. The script first checks whether the input has changed since the last submission before calling the RFG instantiated for the current domain. The output of the RFG is first inspected for parse errors, and whether any special flag is set. Parsing errors are currently handled differently by the client-side component, a general message is shown, because this is an issue that does not has priority in this thesis. However, a different behavior for dealing with parsing errors can easily be plugged into the current implementation. When there are no parsing errors the script checks whether the RFG returns the *Done*-message, in this case the student is asked to solve a new exercise. If the student is not done yet the script sends the raw feedback message from the RFG to the client-side component.

7.3 Conclusion

By combining several techniques from the field of web-development we were able to create a GUI that respects the guide-lines from Section 2.3, and which also resembles the design discussed in Chapter 5. Unfortunately, the round-trip to the server-side increases the response time of the application. By using available libraries in the client-side component we have reduced the amount of round-trips as much as possible. Furthermore, using these libraries made the development of the client-side component easier. The down-side of using the libraries is that the loading time of the application has increased. However, the current implementation is fast enough to be used for testing.

Part IV

Evaluation

Chapter 8

Evaluation

The evaluation of the framework has been done in different ways. To ensure that the framework can indeed be instantiated on different domains, we have made instantiations of the framework on the domains of fractions, logic and linear equations. To show that each of these domains work properly they are accompanied by test-suites showing and verifying their behavior.

Using these instantiations we have made additional test-suites to evaluate the workings of the feedback generation. The results of this evaluation are discussed in Section 8.1. Furthermore, we have asked several teachers and students to evaluate the instantiations using our GUI. The results of these evaluations are given in Section 8.3. Since the algorithm for the fourth phase is already evaluated in the thesis of Hennecke [18], we focus the discussions on the algorithm of the third phase.

Combining the results from both the test-suite evaluation and the comments from the teachers and the students we draw our conclusions in Section 8.4.

8.1 Test suite evaluation

The evaluation in the test-suite consists of lists of scenarios for each domain. Each scenario is tested against a pre-defined set of rules and consists of a PT, a CT and a rule-name. The given rule-name corresponds to the rule that we believe is the result of the algorithm of the third phase given the listed PT and CT. The scenarios are based on our intuition and are meant to test whether the basic functionality works. Therefore, the scenarios for the domain of fractions and the domain of equations only use some of the available operators. It is assumed that the same result can be obtained from rules and terms with other operators. Furthermore, although the listed scenarios use only a limited set of literal values we are certain that different literal values do not change the results.

SameAdd	: A/B + C/B	→	(A+C)/B		
SimpleDiffAdd	: A/B + C/D	→	((A*D) + (C*B))/(D*B)		
ComplexDiffAdd	: A/B + C/D	→	((N*A)+C)/D	where	N := solve(D/B)
SameMin	: A/B - C/B	→	(A-C)/B		
SimpleDiffMin	: A/B - C/D	→	((A*D) - (C*B))/(D*B)		
ComplexDiffMin	: A/B - C/D	→	((N*A)-C)/D	where	N := solve(D/B)
Multiplication	: (A/B) * (C/D)	→	(A*C) / (B*D)		
Division	: (A/B) / (C/D)	→	(A/B) * (D/C)		
SameMixed	: A|B/B	→	A+1		
SolveAdd	: A+B	→	C	where	C := solve(A+B)
SolveMin	: A-B	→	C	where	C := solve(A-B)
SolveMul	: A*B	→	C	where	C := solve(A*B)

Table 8.1: Allowed rules for fractions

PT	*CT*	Expected rule	Actual output
(3/5) / (4/6)	7/9	SolveMul.Division	
5 + 4	8	SolveAdd	
5 * 4	8	SolveMul	
5 - 4	8	SolveMin	
1/2 + 1/2	((1*2) + (1*1))/(2*1)	SimpleDiffAdd	
1/2 + 1/2	((1*1) + (1*1))/(2*1)	SimpleDiffAdd	
1/2 + 1/2	((1*1) + (1*2))/(2*2)	SimpleDiffAdd	
1/5 + 2/7	(1+2)/(5+7)	SameAdd	
3/6+1/5 + 2/5	3/6 + 4/5	SameAdd	
5 + 4 + 7	8 + 7	SolveAdd	
5 - 4 + 7	2 + 7	SolveMin	
1/5 - 2/7	(1-2) / (5-7)	SameMin	
(3/5) * (4/6)	(3*6) / (5*4)	Multiplication	
(3/5) / (4/6)	(3/6) * (6/4)	Division	
(3/5) / (4/6)	(3*5) / (5*4)	Multiplication	
3/5 + 4/6	7/9	SameAdd	SolveAdd
(3/5) * (4/6)	7/9	Multiplication	SolveMul
(3/5) / (4/6)	(3*4)/(30)	Multiplication.Division	SolveMul.Division

Table 8.2: Test scenarios for fractions

8.1.1 Fractions

The scenarios for the domain of fractions are listed in Table 8.2, using the set of rules listed in Table 8.1. Note that the `|`-operator makes it possible to define a mixed fraction. For example, `2|1/2` is rendered as $2\frac{1}{2}$. Furthermore, note that the last column is empty when the actual output is equal to the expected output.

Interesting rules in the set of allowed rules are the rules that actually solve an operator, the rules `SolveMin`, `SolveAdd` and `SolveMul`. These rules are added to the set to allow the algorithm to visit more terms. Without these rules the algorithm can only visit a limited number of terms, because the actual operators cannot be evaluated. Unfortunately, even though the algorithm uses the accuracy measurement in matching, the more generic rules are chosen in some cases. The last three scenarios within Table 8.2 show a few examples of this. The expected rules listed are more accurate and we feel that they should be chosen because the LHS of the rule matches the PT. However, the more generic rules are chosen because the RHS of these rules precisely match the given CT.

Even though the output in the last three cases is not what we expected it to be it can still be justified. After all, the student wants to solve the exercise and using the generic rule for solving the operator is a correct step. Furthermore, the student did not take small steps because the answer was given directly. This violates one of the assumptions of the framework. Therefore, we do not feel that the results of these cases is problematic.

8.2 Logic

Within Table 8.4 the scenarios for the domain of logic are listed, based on the rules from Table 8.3. The scenarios show that most of the basic mistakes are processed correctly by the algorithm. In those scenarios where the result is different from the expected outcome we can see two distinctive cases. The first scenario in which the outcome differs from the expected result shows that recursion does not always lead to better results. We expect to get the rule `ImpElem` from the algorithm, because that is the rule the student must have applied to get the CT. It is a common mistake to forget the $\neg$ in front of the first term during application, therefore this would be the expected outcome. However, the algorithm of the third phase continues to search for a better guess after the application of the rule `ImpElem`, resulting in a guess for `NotNot` because the following rule is generated in the second iteration:

$$\delta : \neg a \rightarrow a$$

Although the output is not expected it can still be justified as being a correct guess.

The last three scenarios are returning an unexpected result because of the large shift in precedence. Within all of the CTs the parentheses around one of the subterms are forgotten. This results in an AST which is heavily skewed, which leads to a large distance between the generated and the desired rule.

EqElem	: A ↔ B	→	(A ∧ B) ∨ (¬(A) ∧ ¬(B))
ImpElem	: A → B	→	¬(A) ∨ B
MorganA	: ¬(A ∧ B)	→	¬(A) ∨ ¬(B)
MorganO	: ¬(A ∨ B)	→	¬(A) ∧ ¬(B)
NotNot	: ¬¬A	→	A
DistAO	: A ∧ (B ∨C)	→	(A ∧ B) ∨ (A ∧ C)
IdemA	: A ∧ A	→	A
IdemO	: A ∨ A	→	A

Table 8.3: Allowed rules for logic

PT	*CT*	Expected rule	Actual output
a∧b	a	IdemA	
¬(a∨b)	¬a∨¬b	MorganO	
¬(a∨¬b)	¬a∧¬b	MorganO	
¬(a∧b)	¬a∨b	MorganA	
¬(a∨b)	¬a∧b	MorganO	
a∨c↔b	(a∨c)∧b∨¬(a∨c)∧¬b	EqElem	
¬a→b	a∨¬b	ImpElem	
¬a→b	a∨¬b	ImpElem	
a→b	a∨b	ImpElem	
a∨b∧c∨¬¬d	a∨b∧c∨¬d	NotNot	
(a∨b)∧(c∨d)	a∨b∧c∨(a∨b∧d)	DistAO	
a∨b∧(c∨d)	a∨b∧c∨(a∨b∧d)	DistAO	
(a∧c)↔b	((a∧c)∧b)∨(¬(a∧c)∧¬b)	EqElem	
(a∧c)↔b	((a∧c)∧b)∨((a∨¬c)∧¬b)	EqElem	
(a∨b)→c	(¬a∧b)∨c	MorganO	
¬¬a↔b	(¬¬a∧b)∨(¬a∧¬b)	EqElem	
a→(b∧c)	a∨(b∧c)	ImpElem	NotNot
¬¬a↔b	(¬a∧b)∨(¬¬¬a∨¬b)	EqElem	MorganA
a∨c↔b	(a∨c∧b)∨¬(a∨c)∧¬b	EqElem	IdemA
a↔b∧(c∨d)	(a↔b∧c)∨(a↔b∧d)	EqElem	DistAO

Table 8.4: Test scenarios for logic

8.2.1 Linear Equations

The last domain that is implemented and tested is the domain of equations. The scenarios for this domain are listed in Table 8.6, based on the rules listed in Table 8.5. Note that the set of rules is adapted to equations, but all of the rules from Table 8.1 could have been added. Furthermore, note that there are no "general" rules within the list of allowed rules. For example, one might expect the following rule to be in the set:

```
Multiply : L=R → L*X = R*X  where X != 0
```

This is not the case because the above rule does not provide the context in which it can be applied. Also, this rule can never be applied because the meta-variable `X` is never instantiated. However, the rule `MulSides` given in Table 8.5 shows the same rule within a context.

The list of scenarios in Table 8.6 again show that the basic mistakes are covered by the algorithm. Also, the failing scenario shows that when a student has both rewritten and evaluated a term the algorithm performs less well. In this scenario, the choice for `DivSidesPlus` is made because there are fewer bindings in this rule, resulting in a lower distance. Although the guess can be justified to some extent, it still shows that the result is only an educated guess.

8.3 Real world tests

The algorithm of the third phase has also been tested by several students and teachers who were familiar with the domain they tested. Before going over their comments we would like to stress that our test-setup is not meant to be of any academic value. It is meant to give a first impression about the workings and the usefulness of such a framework in a real world setting.

Within this real-world setting, each domain has been reviewed by some domain-experts. For the domain of fractions we have asked a primary-school teacher and a graduated student to give their view on the usefulness and correctness of the instance of the framework. The domain of equations is reviewed by a high-school mathematics teacher. Finally, the domain of logic is reviewed by several computer-science grad-students.

The first real-world test was done by a graduated student on the domain of fractions. Since this domain is taught at the end of primary education the knowledge of all the rules was somewhat forgotten, making the reviewing of the tool a refresh course. The results of this test where promising, the returned feedback did help in locating most of the errors, but there was an issue with the phrasing of the message. The complete message was too abstract and only helped because it hinted towards the right solution. After rephrasing the feedback message this domain was reviewed by a primary-school teacher. She also was positive about the received feedback, it would definitely be better than only stating that a step was incorrect, but the feedback message was still a bit

SameAdd	: A/B + C/B	$\rightarrow$	(A+C)/B		
Division	: (A/B) / (C/D)	$\rightarrow$	(A/B) * (D/C)		
PushMulPlus	: A*(B+C)	$\rightarrow$	(A*B) + (A*C)		
PushMulPlusSolve	: A*(B+C)	$\rightarrow$	D + E	where	D := solve(A*B), E := solve(C*A)
DivSidesPlus	: A*B + C = D	$\rightarrow$	B + (C/A) = D/A	where	A != 0
DivSides	: A*B = D	$\rightarrow$	B = D/A	where	A != 0
MulSidesPlus	: (A/B)*x+C = D	$\rightarrow$	A*x + (C*B) = D*(B)		
MulSides	: (A/B)*x = D	$\rightarrow$	A*x = D*(B)		
OtherSideLR	: x + B = D	$\rightarrow$	x = D - B		
OtherSideLRMul	: Ax + B = D	$\rightarrow$	Ax = D - B		

Table 8.5: Allowed rules for equations

PT	*CT*	Expected rule	Actual output
x = 1/2 + 1/2	x = (1+1)/(2+2)	SameAdd	
x = 2(x + 1)	x = 2x + 1	PushMulPlus	
2(x + 1) = 4	x + (2*1) = 4	PushMulPlus	
2(x + 1) = 4	x + 2 = 4	PushMulPlusSolve	
(1/2)*x = 2	x = 2	DivSides	
0*x = 5	x = 5/0	DivSides	
2x + 4 = 8	x + 4 = 8/2	DivSidesPlus	
x + 4 = 6	x = 6 + 4	OtherSideLR	
x + 4 = 6	x = 6 + 4	OtherSideLR	
(1/2)*x+4= 8	1*x + 4 = 8*2	MulSidesPlus	
(1/2)*x+4= 8	1*x + 4 = 16	MulSidesPlus	
(1/2)*x+4= 8	x + 4 = 8*2	MulSidesPlus	
(1/2)*x+4= 8	x + 4 = 16	MulSidesPlus	DivSidesPlus

Table 8.6: Test scenarios for equations

abstract. Intellectually gifted students would enjoy the feedback because they can figure it out, but lesser gifted students might get confused by the messages.

During the same time as the review of the domain of fractions, several grad-students reviewed the domain of logic. These students have all learned the skill of rewriting logical expressions during their first year and were asked whether this tool would have been of any help. The students where positive about the tool and the received feedback. Even though it was clear to them that the framework only made an educated guess they confirmed our believe that the feedback is indeed helpful. Furthermore, there where some compliments about the workings of the GUI.

The last real-world test is a review of the domain of linear equations by a high-school mathematics teacher. By using the knowledge of usual mistakes that are made by students she quickly found that the set of rules defined for the domain of linear equations did not define enough rules. Or rather, the set of rules did not contain the rules that encoded the correct behavior for the mistakes that are usually made. This led to a higher number of incorrect guesses. Since it was known that the rule only made an educated guess, this was not a big problem, although it would be nice to improve on this. One other thing that was addressed in the evaluation of the domain was again the phrasing of the feedback message. Even though it is understandable for some students it would probably scare students away because of its abstract nature. A suggestion was made to add a more concrete example of the rules, either in the message or in a separate text-area, to make the feedback more concrete for the students.

8.4 Conclusion

Given the results of both the test-scenarios and the real-world tests we believe that the third phase algorithm is able to cover the basic scenario's of the different domains. Furthermore, we can draw the following conclusions about the third-phase.

First, the problem of distance created by a large shift in priority is noticeable in one domain, but it does not seem to have much influence on the other two. Furthermore, within the domain in which it was noticeable it only shows up in scenarios involving a single rule. Therefore we believe that shifts in priority have only a minor impact on the workings of the algorithm.

The second conclusion that can be made is that the domain of fractions might not be a good domain for the framework. Even though the domain can be seen as a rewrite system this view does not fit with how fractions are currently taught in primary education, a violation of the rules of Beeson. On the other hand, it might be possible to create a feedback-message which is based on rewrite-rules without revealing this property. In that case the framework can be used in the current curriculum and the rules of Beeson would be respected.

A third conclusion is that it is not a large problem that the algorithm of the third phase always returns a rule, although preventing the algorithm from making completely wrong guesses would be nice. This conclusion is based on the fact that the students can be told that this phase only makes an "educated guess" and place the feedback in the right perspective. Furthermore, in general, the set of allowed rules for a domain is rather small so we imagine that all of the allowed rules are available. When this is not the case the algorithm performs less well, showing a trade-off between effort to implement and the quality of feedback.

As a fourth conclusion we can see that more accurate rules produce more accurate result. For example, within the rules for linear equations we have seen the rule `PushMulPlusSolve` which encodes both rewrite as well as solve steps. Having this rule in the rule-set leads towards better results. We can imagine that these kind of rules can be generated by applying a set of rules that solve operators to the set of actual rewrite rules. Using this generated rule-set eliminates the need for having rules that solve operators. This also solves the problem of returning rules that are too general we have encountered within the domain of fractions.

Lastly, we can see that the number of returned merged rules is rather low. There is a total number of two merged rules in all of the test-suites for the three domains. Furthermore, only a few feedback-messages contained a merged rule during the real-world tests. These rules where noticed, but they where not seen as "better". This leads to the conclusion that extending the rule-set is not a real addition to the quality of the system. On the other hand, the generation of the merged rules is rather cheap and the merged rules are not seen as useless. Therefore, we intend to leave this functionality within the framework, maybe making it optional.

The overall conclusion of this evaluation is that using the third phase alone is always a compromise requiring little work to create the desired input and receiving reasonable feedback in a large number of cases. By combining the third and fourth phase the quality of the feedback is in the hands of the implementer because it is directly related to the effort that is put into the configuration.

Chapter 9

Future Work

Though the evaluation of the framework is promising, there are still several things that can be improved. There is future work available in three area's considering this thesis: the third phase algorithm, the RFG-framework as a whole and the GUI.

9.1 Third phase algorithm

Even though the evaluation of the third phase showed that the shift of priorities is not immediately problematic, we believe that this issue can be solved. We would suggest a new algorithm that addresses large changes in priorities by re-shaping the AST of a term. This re-shaping can either be done by pushing down operator nodes, or adding parenthesis around subterms within the textual representation. This last approach would require a set of parser combinators with some sort of priority knowledge. Both of these approaches are subject of future research, although we can also imagine that it is easier to generate buggy rules for these kind of scenario's.

A second improvement to the third-phase is a filtering function for guessed rules. The current state of the guesses is certainly useful, but there is a change that the guess is off, or even incorrect. Even though students can be made aware of the fact that the algorithm gives only an educated guess, incorrect guesses might confuse a student. How a guess can be filtered on relevancy in a generic way should be part of empirical research.

The last improvement must also be subject of empirical research: improving the phrasing of the feedback message of this phase. Although the message can contain a variety of information, it may all be ignored when the phrasing is too abstract or is not on the level of the intended audience. Designing a feedback message with enough information, and on the correct level for the targeted audience can be quite hard. A research aimed at defining guidelines for such a message can help in improving the overall quality of the generated feedback.

9.2 The Framework

Future work for the framework consists of some practical enhancements. First, the configuration of the tool is now captured in a separate module. It would be more convenient, and make the tools more adaptable, if this configuration is read from a external file. Second, there are several upper-bounds mentioned in Section 4.7 which can still be implemented. These are not available in the current implementation because the instantiated tools where fast enough for testing purposes.

A more research related question is that of the merging of the rule-set. When rules are merged it could be that, because of a mistake of the implementer, the assumption that each rule rewrites to the answer is violated. This results in an infinite loop during the merging of the rule-set. Detecting and reporting such an infinite loop would help in creating correct sets of rewrite rules.

Lastly, the quality of the sets of rewrite rules can be improved by making it easier to specify classes of rewrite rules. This can, for example, be done by extending the syntax with meta-operators. Furthermore, generating new rules by applying rules for solving operators to a certain set of rules might be a useful addition. We imagine that the quantity and quality of rule-sets will increase when the definition of rules is made easier.

9.3 The GUI

Although the feedback on the GUI was positive it is not to be used as a full-blown educational tool in its current implementation. There is a lack of all kind of administrative features ranging from remembering class-names to keeping a history of each student. This does not pose a problem for this thesis because it was not an objective. We do believe that the current implementation can serve as a basis for further development. Furthermore, using only the graphical view does not give a good user-experience, because it is not possible to visually select (part of) a term. We can imagine that this kind of visual select in the context of web-applications is a research subject on its own.

Chapter 10

Conclusion

Within this thesis we have given an overview of the quality of generated rule-feedback in current educational tools. We showed that even though it is possible to generate high-quality feedback, most tools currently available do not seem to rise above the level of *there was a mistake*. The tools that are capable of generating high-quality feedback are the result of many years of research. Transferring the results of this research to other domains is of no academic value because the research is already done. On the other hand, commercial implementers of educational tools seem to be unaware of these possibilities. It could also be the case that they believe that the trade-off between expected profit and implementation effort is not interesting. Either way, the current quality of feedback in educational tools needs to be improved.

In order to address this problem we have developed a generic framework that can be instantiated with little effort. The quality of the feedback that is generated by the instantiated framework depends on the amount of work that is put into the definition of the different sets of rules. When no rule is defined the instantiated framework generates feedback on the lowest level, comparable to the level of many tools that are currently available. Putting effort into defining sets of allowed-rules or buggy-rules allows the instantiated framework to generate high-quality feedback. This allows for a trade-off between the amount of work done and the quality of the generated feedback.

Even though the algorithm for the third phase can be improved, it has quit some potential and is already of practical use. By combining this algorithm with an existing algorithm [18], the current short-comings can be solved in an easy manner. Because of the flexible nature and incremental configuration we believe that the framework is a valuable addition to the current range of educational tools.

Although the current implementation has some rough edges, we believe it provides a good reference implementation, and an interesting research subject. Even though the usefulness of the generated feedback still has to be validated by empirical research our conversations with experts from different domains already show that the framework has a great potential.

Part V

Appendixes

Appendix A

Rewrite rules

This appendix introduces the syntax and the semantics of the rewrite rules implemented within the framework. Although the syntax is specific to a particular domain, there are several generic constructs. These constructs are discusses in Section A.1. Furthermore, after parsing the rules several checks are performed to verify the sanity of the rules. We explain these checks and their purpose in Section A.2.

A.1 Syntax and semantics

The syntax of a rule is given in Figure A.1. On the top-level a rule consists of a name and two terms followed by an optional list of restrictions. A name is one or more characters, either upper-case or lower-case, and is used to identify a rule.

The syntax of the terms in the rule depends heavily on the syntax of the domain and the supplied parser. Since the domain dictates the different operators and literals it cannot be generically defined. Moreover, even though the supplied parser must be supplied with a parser for meta-variables, it is up to the implementer to decide where these meta-variables may be parsed. The only thing that is defined in this implementation is that a single upper-case character is parsed as a meta-variable.

There are currently three possible restrictions 1) a term must be equal to a second term (`Equal`), 2) a term is equal to a solved term (`Solve`) and 3) a term is not equal to a second term (`InEqual`). One thing that is important for the ordering of restrictions is that they are evaluated left-to-right. This means that a restriction may only use new variables, introduced by for example a `solve`-restriction, that are defined on its left. When this ordering is not respected the application of the rule fails.

Application of a rule is done by first matching the LHS of a rule against a term in a top-down algorithm. A match only succeeds when the top-level constructors are equal and each child can be matched. The only exception is the case in which a meta-variable is matched against a sub-tree. When the meta-variable is not previously bound the match succeeds and creates a binding, otherwise the match fails. Given a successful match, the restrictions belonging to the rules, if any, are collected and checked using

```
Rule = Name ":" Term "=>" Term ("where " Restrictions)?

Term = domain dependent, but includes MetaVar

MetaVar = [A-Z]+

Name = [A-Za-z]+

Restrictions = Restriction "," Restrictions
             | Restriction

Restriction  = Equal | InEqual | Solve

Equal   = Term ":=" Term
InEqual = Term "!=" Term
Solve   = Term ":=" "solve(" Term ")"
```

Figure A.1: Syntax of rules in BNF

the evaluation order and semantics described above. When all is well, the RHS term is instantiated on the bound variables resulting in a new term.

A.2 Checks

Before a rule is used, there are four checks performed to see whether the rule can actually be applied. The first check is whether all the variables on the RHS of the rule either appear in the LHS or are instantiated by a solve-restriction. When this is not the case the RHS could never be built because at least one meta-variable is never bound.

The second check makes sure that the LHS of each restriction is a meta-variable. This does not only make the implementation easier, it also makes the restrictions more intuitive to read and understand. Keep in mind that the rules might be read by laymen, so they must be relatively easy to understand.

The third restriction checks whether each variable is only mentioned once on the LHS of a restriction. Actually, the implementation allows a meta-variable to be mentioned twice or more but makes sure that the RHS of each of these restrictions is exactly equal. Our first implementation allowed the RHS of restrictions with an equal LHS to be *semantically* equal, but this caused problems during the merging of the rule-set.

The fourth and last restriction checks whether each restriction does not depend on itself. This is also done to make the restrictions more intuitive to read and understand.

Bibliography

[1] M. J. Beeson. A computerized environment for learning algebra, trigonometry, and calculus, 1990.

[2] M. J. Beeson. Design principles of mathpert: Software to support education in algebra and calculus. In *Computer-Human Interaction in Symbolic Computation*. 1998.

[3] C. Bokhove, A. Heck, and G. Koolstra. Intelligent feedback to digital assessments and exercises (in dutch). *Euclides*, 2005.

[4] J. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 1980.

[5] J. S. Brown and R. B. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 1978.

[6] A. Bundy. *The computer modelling of mathematical reasoning*. Academic Press Professional, Inc., 1985.

[7] R. B. Burton. Diagnosing bugs in a simple procedural skill. In *Intelligent Tutoring Systems*. 1982.

[8] W. Castelo-Branco Lins. Educational software interfaces and teacher's use.

[9] H. Chaachoua *et al.*. Aplusix, a learning environment for algebra, actual use and benefits. 2004. Retrieved from http://www.itd.cnr.it/telma/papers.php, January 2007.

[10] P. A. Chalmers. User interface improvements in computer-assisted instruction, the challenge. 2000.

[11] A. M. Cohen, H. Cuypers, D. Jibetean, and M. Spanbroek. Interactive learning and mathematical calculus. In *MKM*, 2005.

[12] I. Erev, A. Luria, and A. Erev. On the effect of immediate feedback. Retrieved from http://telem-pub.openu.ac.il/users/chais/2006/05/pdf/e-chais-erev.pdf, December 2006.

[13] S. Peyton Jones *et al.*. *Haskell 98, Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[14] S. Fowler and V. Stanwick. *Web Application Design Handbook*. Elsevier Inc., 2004.

[15] Freudenthal Institute. Digital math environment. http://www.fi.uu.nl/dwo, 2004.

[16] D. Frye and E. Soloway. Interface design: a neglected issue in educational software. In *CHI '87: Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, 1987.

[17] G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. *Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences Sharing. Good Practices of Research Experimentation and Innovation.*, 2005.

[18] M. Hennecke. *Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen (in German)*. PhD thesis, Hildesheim University, 1999.

[19] J. E. Hinostroza and H. Mellar. Pedagogy embedded in educational software design: Report of a case study. *Computers and Education*, 2001.

[20] Steve Krug. *Don't Make Me Think: A Common Sense Approach to Web Usability*. New Riders Press, 2000.

[21] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Department of Computer Science, Universiteit Utrecht, 2001.

[22] W. C. Lins. Educational software interfaces and teacher's use.

[23] J. Lodder, J. Jeuring, and H. Passier. An interactive tool for manipulating logical formulae. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2006.

[24] M. Marvrikis and A. Macioncia. Wallis: a web-based (ile) for science and engineering students studying mathematics. In *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, 2006.

[25] B. Mason and R. Bruning. Providing feedback in computer-based instruction: What the research tells us. retrieved from http://dwb.unl.edu/Edit/MB/MasonBruning.html, august 2006, 2001.

[26] Ministerie OCW. Actieplan verbonden met ict (in dutch). 2006. Retrieved from http://www.minocw.nl/ict/documenten/index.html, January 2007.

[27] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. *Haskell Workshop 2007*, 2007.

[28] E. H. Mory. Feedback research revisited. In *Handbook of Research on Educational Communications and Technology*. 2004.

[29] H. Passier and J. Jeuring. Ontology based feedback generation in design-oriented e-learning systems. In *Proceedings of the IADIS International conference, e-Society*, 2004.

[30] H. Passier and J. Jeuring. Feedback in an interactive equation solver. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2006.

[31] R. Raymond *et al.*. Successful pedagogical applications of symbolic computation. *Computer-Human Interaction in Symbolic Computation*, 1999.

[32] C. D. Ryan. The human-computer interface: challenges for educational multimedia and web designers. *SIGCSE Bull.*, 2001.

[33] Sangwin, J. Christopher, and M. Grove. Stack: addressing the needs of the "neglected learners". In *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, 2006.

[34] B. Shneiderman. *Designing the user interface (2nd ed.): strategies for effective human-computer interaction*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[35] J. Zuidema and L. van der Gaag. *De volgende opgave van de computer*. OMI Universiteit Utrecht, 1993.