Using Constraint Logic Programming to Generate Drills in Mathematics*

Ana Paula Tomás, José Paulo Leal, and Pedro Vasconcelos

DCC-FC & LIACC, Universidade do Porto,
R. do Campo Alegre, 823, 4150-180 Porto, Portugal
Fax: +351 22 6003654 & Phone: +351 22 6078830
{apt,zp,pbv}@ncc.up.pt

Keywords: Automatic Generation of Drills, Computer-based training, Constraint Logic Programming, AI Foundations and Knowledge Representation.

Abstract. We propose a methodology for designing online exercises systems with special focus on applications to Mathematics education. The major goal is to develop a web-based environment that make available exercises and solutions to students and teachers. Promising results are reported in this paper that suggest that Constraint Logic Programming frameworks are adequate to implement such a system. These languages have the right expressiveness to encode control on the system in an elegant and declarative way.

1 Motivation

Not all students have high mathematical skills but surely one of the reasons for the lack of success in mathematics is that too often students merely memorize how to solve some exercises, instead of trying to understand the fundamental concepts and results. Hence, a possible drawback of classical textbooks and some existing online course-ware and exercise systems is that the proposed problems are quite pre-defined, either fixed or at best randomly generated instances of the same problem template [5, 6].

Rather than to reproduce the classical textbooks, advances in the computer technology and the Internet should be exploited to develop really interactive and re-usable contents. Quite sophisticated web-based learning environments are being developed. For example, ActiveMath [16], that is a second-generation interactive textbook project offering user-adaptiveness and re-usability by employing an `XML`-based representation of mathematical knowledge and Artificial Intelligence techniques. Alike [5], it supports exploratory learning through communication with mathematical systems.

Commercial mathematical systems, as *Geometer's Sketchpad* [7], *Maple*[12] and *Mathematica* [14], just to name a few, are often used as mathematics tools

* Work partially supported by funds granted to LIACC through *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*.

for exploratory learning [9, 17], enabling the students to try their own examples. Some already offer access to their applications through web browsers [14]. The focus of this paper is not on problem solving in the broader sense of exploration, but rather on the repetitive drills students have to do for consolidation of concepts and practice of algebraic procedures. For constructive learning to be effective, students need self-confidence and also basic knowledge.

Lots of mathematics teaching resources are spread over the Web, namely web-based systems for computer aided training and/or assessment, with authoring facilities for teachers to create question files, for example, for homework and assignments (e.g. [2, 6, 10, 16]). This requires non-negligible effort from the teacher, specially to generate problem instances that are not immediately recognized as simple variants of a few basic expressions. In fact, for all the systems we know, the exercises are not generic enough and the user can almost anticipate the form of the next instance of the problem, after a while.

This paper reports on our experiences in using a computer algebra system (*Maple*) and Constraint Logic Programming (CLP) frameworks to automatically generate online examples and exercises for teaching and learning a topic in mathematics. Our final goal is to develop a system that dynamically computes a wide range of examples that really look different for students, despite they naturally obey some given specification. Moreover, the system shall optionally provide explanations that may help students improve their ability to express coherently in mathematical language.

Our approach has many different potentials that include user-adaptiveness, easy definition of several curricula, and possible integration in intelligent tutoring systems. The methodology we adopt to characterize an application domain mimics the one teachers usually follow when they try to formulate basic problems in some context. First, we have to define and represent the forms of exercises that may be solved by the procedures that students shall learn. Second, the exercises must have pedagogical interest, so that we must have some idea of their solutions. The best strategy is then to proceed from an intended solution and/or solving procedure to formulate a problem instance. In this way, we may also ensure that the generated problems are solvable by the computational system (and, hopefully, by the student at a given level), thus avoiding undecidability issues. Although not all the topics taught in mathematics at high school allow such an automatic treatment, a considerable number do. Many of the questions that students have to work out in Mathematics courses may successfully be solved by algebraic procedures. If these procedures are also implemented, the generated problems can be completely solved and the solving steps explained, in contrast to systems that implement deduction schemes, as theorem provers.

This idea is also implicit in a recent work by Sangwin [18], although the emphasis there is on how to generate exercises that get the students to construct instances of mathematical objects with some properties. How to reduce the teachers' effort to prepare questions is not considered at all and, moreover, it is assumed that they have some expertise in writing computer programs. Actually, it is examined the application of *AIM* [10], which is an authoring system for

computer aided assessment that ultimately uses *Maple* to process the exercises but that counts on the teacher to program the exercises and in some situations their grade scheme. This is quite different from what we have in mind.

Although we expected that the use of computer algebra systems could highly reduce our implementation effort, our experience (with *Maple*) has shown that the algebraic simplifications may turn out troublesome. As we shall detail in the following section, some additional constraints shall be imposed, for instance, on the expressions that arise in the exercises, to avoid inconsistencies in the explanations that are produced.

This paper is intended to present the results we have achieved so far, and extends [23]. To illustrate the main ideas we refer to a particular topic in Introductory Calculus, giving, in Section 3, a grammar that characterizes a vast sample of examples from some high school textbooks. This grammar shall still be extended to cover other functions taught. The interesting point is that we now may get specialized forms of the expressions almost for free, by adding further restrictions through constraints. This is of great importance for educational purposes since the system must be parametrized to easily cater for different curricula. In the following section, we discuss strengths and weaknesses of our first attempt to implement some programs to generate problems and examples using a computer algebra environment. The need for a more declarative framework, led us to investigate the use of CLP, that offers natural support for possible user-defined constraints on the expressions. Sections 4 and 5 describe aspects of our prototype implementation. Finally, we also briefly address interface issues and conclude. Programs that have been developed as a test-bed for some of the ideas may be download from `http://www.ncc.up.pt/~apt/demomath.html`.

2 Some Experiments Using a Computer Algebra System

In this section, we discuss some pros and cons of using *Maple* to develop interactive course-ware, which may be common to other computer algebra systems. Our previous work involved the design of *Maple* worksheets to present some specific topic in mathematics. Besides some concise notes on the addressed issue, such worksheets typically include pointers to other ones where the end-user student may find randomly generated examples and exercises to work on.

Some of the algebraic procedures that students learn are crucial to different problems. For example, in introductory calculus, the analysis of the sign variation, zeros and domain of real-valued functions is a basic tool to find intervals where a function is monotonic, to study concavity and convexity for twice differentiable functions and to sketch their graphs. But, it is quite easy to define functions for which no generic algorithm exists to accurately compute their zeros, as shown independently by Abel and Galois, about a century ago, for polynomials of degree greater than four. In the following example, we consider the generation of rational functions defined by a quotient of two polynomial

expressions so different as the following ones.

$$\frac{(-x-\sqrt{5})^5}{x^5-4x^4+6x^3} \qquad \frac{(-x^3-x^2)}{(x+2)^3} \qquad \frac{(x+1)^4(x^5+3x^4+2x^3)}{(x^2+1)^2(-2x-2-\sqrt{7})}$$

Example 1. We shall present some output from one of our *Maple* programs that is helpful for the discussion. Some typesetting has in fact been done for space reasons.

```
 > domains(true);
FIND THE DOMAIN OF THE FUNCTION f DEFINED BY
```

$$f(x) = \frac{(8x^2+14x-15)(2x+1)}{(4x^6-x^5-5x^4)(3x^2-17x+10)^2}$$

```
SOLUTION: Being f a rational function, it is defined for all
real numbers except the zeros of the denominator of its expression.
We have
```

$$(4x^6-x^5-5x^4)(3x^2-17x+10)^2=0$$

```
if and only if
```

$4x^6-x^5-5x^4=0$ `or` $(3x^2-17x+10)^2=0$.

```
As concerns
```

$4x^6-x^5-5x^4=0$, `we have`

$$\begin{aligned} 4x^6-x^5-5x^4=0 &\Leftrightarrow x^4(4x^2-x-5)=0 \\ &\Leftrightarrow x=0 \vee 4x^2-x-5=0 \end{aligned}$$

```
To solve
```

$4x^2-x-5=0$, `we apply the solving formula for polynomial equations of degree 2, the roots being -1 and 5/4.`

```
As concerns
```

$(3x^2-17x+10)^2=0$, `we have`

$$(3x^2-17x+10)^2=0 \Leftrightarrow 3x^2-17x+10=0$$

```
To solve
```

$3x^2-17x+10=0$, `we apply the solving formula for polynomial equations of degree 2, the roots being 2/3 and 5.`
`We conclude that all real numbers are in the domain of f, but 2/3, 0, -1, 5 and 5/4.`

As other computer algebra systems, *Maple* supports polynomial expressions and thus it is easy to implement this procedure. The main issue is how to control the difficulty level of the problems. The idea is to abstract their form from the rules students could use to solve them, as we mentioned previously. For educational purposes, we also need to have some control on the generated polynomial expressions, so that the exercise may have pedagogical interest.

Instead of simply using the builtin *Maple* procedure to generate random polynomials, the computation of $f(x)$ was driven by the selection of the set of roots, which might be rational and (conjugated) irrational numbers. Factors with no real roots were obtained by adding appropriate constants to quadratic polynomial expressions with real roots to shift their representing parabolas upwards

or downwards so that every intersection with the horizontal axis is eliminated. Both the denominator and numerator are factored and the factors may be of the following basic forms: $ax + b$, $ax^2 + bx + c$, $(ax + b)^n$, $(ax^2 + bx + c)^n$ and $ax^{n+1} + bx^n$, $ax^{n+2} + bx^{n+1} + cx^n$, where $a, b, c \neq 0$. The idea is that the student has to know how to solve linear and quadratic equations, $U^n = 0$ and that $ax^{n+1} + bx^n = x^n(ax + b)$ and $ax^{n+2} + bx^{n+1} + cx^n = x^n(ax^2 + bx + c)$. In this phase, we discarded expressions as $(ax^{n+2} + bx^{n+1} + cx^n)^m$, because we did not think they are of great pedagogical interest.

An important point that deserves some further research is how to improve the linguistic quality of the output explanations. It is not trivial to obtain explanations in natural language by annotating the programs. In Example 1, almost no use was made of global context information, which renders the explanations fairly repetitive and, therefore, unnatural or pedagogically poor. This observation applies also to intelligent tutoring systems.

Besides the need for a more flexible input/output interface, three other remarks have played a major role in our decision to try a different platform. The first one concerns the algebraic manipulations that *Maple* automatically performs, which may result in unpredictable simplifications of the expressions being operated. This feature appears as a great advantage when compared to Logic Programming systems but is surely a serious drawback for our intended usage of the system. Actually, it may introduce some puzzling inconsistencies in the output explanations. For instance, it is not possible to pretty print $3(x^2 + 5)$ in *Maple* since it will naturally yield $3x^2 + 15$. By a similar reason, we had better not ask the student to find the domain of a rational function defined by $f(x) = (x-1)^2/(x-1)$, because that expression would be printed as $f(x) = x-1$, and hence 1 belongs to domain of the latter but not of the former one. Computer algebra systems like *Maple* and *Mathematica* do some basic two-dimensional formatting of the output presented to the user, but they have some limitations: the automatic re-ordering and simplification routines might modify the expression presented; they also have limited text formating possibilities (for example: mixing formulas and text in a single line, or formatting tables).

In Example 1, the simplifications were avoided by further restricting the types of the generated rational functions $f(x)$ to disallow repetitions of factors (either in a product or quotient) and to require that the involved polynomials just have integer coefficients. Since we would like to cover more general expressions, this does not seem the right way to proceed.

The second point is that we need declarativeness to help specify the possible form of the expressions and problem templates. Finally, we would like the application to be well parametrized to cater for different curricula. For both these aspects, CLP seems to offer the right expressiveness to encode control on the system in an elegant way. The disadvantage is that we now have to implement symbolic processing of algebraic expressions to provide exact representations of the solutions, which hopefully have quite simple pre-defined forms, as we further detail in Section 5. It is worth mentioning that CLP languages are rather adequate for symbolic processing. All one needs is either to spend sometime im-

plementing a symbolic processor or to find and adapt an existing one. Difficulties have also appeared when we tried to combine different constraint solvers, since it is almost impossible to share variables between them in a natural way.

We shall now present abstract representations for the expressions, that we need to characterize the problem templates and to simplify the solving procedures. For that purpose, we give a grammar that characterizes a wide range of the function expressions that may be found in high school textbooks and whose zeros can be exactly computed. This grammar extends the set of functions we considered in Example 1.

3 Using Grammars and Constraints to Define Expressions

In order to be able to abstract the possible forms of function expressions, we have carried out a thorough analysis of Portuguese textbooks in mathematics for grades 10 to 12. As a result, we defined the grammar shown in Fig. 1.

For prototyping, the trigonometric, exponential and logarithmic functions have been left out. Basically, with this grammar we try to capture some of the expressions for which the computation of the domain and zeros mainly involves solving linear or quadratic equations ($ax+b=0$ or $ax^2+bx+c=0$), or equations of the form $aX^n+b=0$, $a\sqrt[n]{X}+b=0$, $X^n \pm Y^n=0$, $\sqrt[n]{X} \pm \sqrt[n]{Y}=0$, for $n \geq 2$, or $X/Y \pm Z/T=0$, with $degree(XT) \leq 2$ and $degree(YZ) \leq 2$, or even some case-based reasoning to get rid of the absolute value operators. We note that by writing, for instance, $(\texttt{k*})^?\texttt{rad(}basic_{12}, N\texttt{)} + (\texttt{k*})^?\texttt{rad(}basic_{12}, N\texttt{)}$ we really want to restrict N to be the same for both subterms, so that the grammar is not context-free[1]. We use $(\texttt{k*})^?\texttt{rad(}basic_{12}, N\texttt{)}$ as an abbreviation for $\texttt{k*rad(}basic_{12}, N\texttt{)}$ or $\texttt{rad(}basic_{12}, N\texttt{)}$, and * means product.

The rightmost column of following table contains the output expressions that correspond to the basic types. The first two columns contain the internal representations we use to denote them. Two levels of abstraction are considered.

p1 o $TypeT$	pol(T, $[a,b]$)	$aT+b$
p2 o $TypeT$	pol(T, $[a,b,c]$)	aT^2+bT+c
xip$(1,N)$	expand(N, x, pol(x, $[a,b]$))	$ax^{N+1}+bx^N$
xip$(2,N)$	expand(N, x, pol(x, $[a,b,c]$))	$ax^{N+2}+bx^{N+1}+cx^N$
pow(N) o $TypeT$	pow(T, N)	T^N
rad(N) o $TypeT$	rad(T, N)	$\sqrt[N]{T}$
abs o $TypeT$	abs(T)	$\lvert T \rvert$
p2 o pow(N) o x instead of bisqr(N)	pol(pow(x,N), $[a,b,c]$) instead of $bisqr$	$ax^{2N}+bx^N+c$

It may be checked that

$$\frac{(8x^2+14x-15)(2x+1)}{(4x^6-x^5-5x^4)(3x^2-17x+10)^2}$$

[1] Indeed, it is known that $\{0^n10^n10^n \mid n \geq 1\}$ is not a context-free language.

$$
\begin{array}{lll}
function & \longrightarrow & (\texttt{k*})^{?}prodfact \mid (\texttt{k*})^{?}divexpr \\
prodfact & \longrightarrow & factor \mid prodsexpr \\
divexpr & \longrightarrow & prodfact\texttt{/}prodfact \mid \texttt{k/}prodfact \mid prodfact\texttt{/k} \\
 & \longrightarrow & \texttt{pow(}divexpr, N\texttt{)} \mid \texttt{rad(}divexpr, N\texttt{)} \mid \texttt{abs(}divexpr\texttt{)} \\
prodexpr & \longrightarrow & factor\texttt{*}factor \mid factor\texttt{*}prodexpr \\
 & \longrightarrow & \texttt{pow(}prodsexpr, N\texttt{)} \mid \texttt{rad(}prodsexpr, N\texttt{)} \mid \texttt{abs(}prodsexpr\texttt{)} \\
factor & \longrightarrow & sumexpr \mid vxip \mid basic \\
sumexpr & \longrightarrow & \texttt{abs(}sumexpr\texttt{)} \mid \texttt{pow(}sumexpr, N\texttt{)} \mid \texttt{rad(}sumexpr, N\texttt{)} \mid bsum \\
bsum & \longrightarrow & ipol_1(vquot_{12k}) \\
 & \longrightarrow & (\texttt{k*})^{?}\texttt{rad(}basic_{12}, N\texttt{)} + (\texttt{k*})^{?}\texttt{rad(}basic_{12}, N\texttt{)} \\
 & \longrightarrow & (\texttt{k*})^{?}\texttt{pow(}basic_{12}, N\texttt{)} + (\texttt{k*})^{?}\texttt{pow(}basic_{12}, N\texttt{)} \\
 & \longrightarrow & (\texttt{k*})^{?}\texttt{pow(}basic_{12}, N\texttt{)} + (\texttt{k*})^{?}\texttt{pow(}basic_{1}, 2N\texttt{)} \\
 & \longrightarrow & (\texttt{k*})^{?}\texttt{rad(}basic_{12}, 2N\texttt{)} + (\texttt{k*})^{?}\texttt{rad(}basic_{1}, N\texttt{)} \\
 & \longrightarrow & (\texttt{k*})^{?}\texttt{rad(}2, basic_{12}\texttt{)} + (\texttt{k*})^{?}basic_1 \\
 & \longrightarrow & (\texttt{k*})^{?}\texttt{pow(}2, basic_{1}\texttt{)} + (\texttt{k*})^{?}basic_{12} \\
 & \longrightarrow & (\texttt{k*})^{?}basic_{12} + (\texttt{k*})^{?}basic_{12} \\
 & \longrightarrow & (\texttt{k*})^{?}quot_{12k} + (\texttt{k*})^{?}basic_{12}, \text{ subject to } \textbf{Condition} \\
 & \longrightarrow & (\texttt{k*})^{?}quot_{12k} + (\texttt{k*})^{?}quot_{12k}, \text{ subject to } \textbf{Condition} \\
vquot_{12k} & \longrightarrow & \texttt{pow(}vquot_{12k}, N\texttt{)} \mid \texttt{rad(}vquot_{12k}, N\texttt{)} \mid quot_{12k} \\
quot_{12k} & \longrightarrow & \texttt{k/}basic_{12} \mid basic_{12}\texttt{/k} \mid basic_{12}\texttt{/}basic_{12} \mid \texttt{abs(}quot_{12k}\texttt{)} \\
basic_{12} & \longrightarrow & basic_1 \mid basic_2 \\
basic_2 & \longrightarrow & fpol_1(\texttt{abs(}basic_2\texttt{)}) \mid ipol_2(\texttt{x}) \mid \texttt{expand(}1, \texttt{x}, ipol_1(\texttt{x})\texttt{)} \\
 & \longrightarrow & basic_1\texttt{*}basic_1 \mid fpol_1(\texttt{pow(}2, basic_1\texttt{)}) \mid \texttt{pow(}2, basic_1\texttt{)} \\
 & \longrightarrow & \texttt{abs(}basic_2\texttt{)} \\
basic_1 & \longrightarrow & \texttt{abs(}basic_1\texttt{)} \mid fpol_1(\texttt{abs(}basic_1\texttt{)}) \mid fpol_1(\texttt{x}) \\
basic & \longrightarrow & ipol_2(\texttt{x}) \mid \texttt{expand(}1, \texttt{x}, ipol_1(\texttt{x})\texttt{)} \mid bisqr \mid fbasic \\
 & \longrightarrow & fpol_1(fbasic) \mid fpol_1(\texttt{x}) \\
fbasic & \longrightarrow & \texttt{abs(}basic\texttt{)} \mid \texttt{pow(}basic, N\texttt{)} \mid \texttt{rad(}basic, N\texttt{)}, N \geq 2 \\
vxip & \longrightarrow & xip \mid \texttt{k*}vxip \mid \texttt{abs(}vxip\texttt{)} \mid \texttt{pow(}vxip, N\texttt{)} \mid \texttt{rad(}vxip, N\texttt{)}, N \geq 2 \\
xip & \longrightarrow & \texttt{expand(}N\texttt{,x,}ipol_2(\texttt{x})\texttt{)} \mid \texttt{expand(}N+1\texttt{,x,}ipol_1(\texttt{x})\texttt{)}, \quad N \geq 1 \\
bisqr & \longrightarrow & ipol_2(\texttt{pow(x,}N\texttt{)}), \quad N \geq 2 \\
fpol_1(T) & \longrightarrow & \texttt{pol(}T\texttt{,[}a, b\texttt{])}, \quad a \neq 0 \\
ipol_2(T) & \longrightarrow & \texttt{pol(}T\texttt{,[}a, b, c\texttt{])}, \quad abc \neq 0 \\
ipol_1(T) & \longrightarrow & \texttt{pol(}T\texttt{,[}a, b\texttt{])}, \quad ab \neq 0 \\
\texttt{x} & \longrightarrow & variable \\
\texttt{k} & \longrightarrow & constant
\end{array}
$$

Condition: Being either of the form $(\texttt{k*})^{?}A/B + (\texttt{k*})^{?}C$ with $degree(BC) \leq 2$ or of the form $(\texttt{k*})^{?}A/B + (\texttt{k*})^{?}C/D$ with $degree(AD) \leq 2$ and $degree(BC) \leq 2$.

Fig. 1. Describing functions that may appear in exercises

is of the form

$$\frac{\texttt{pol}(x,[8,14,-15]) * \texttt{pol}(x,[2,1])}{\texttt{expand}(4,x,\texttt{pol}(x,[4,-1,5])) * \texttt{pow}(2,\texttt{pol}(x,[3,-17,10]))}$$

And, we may also conclude that e.g., $-2|-2y+4|+4|3y+3|+2$ belongs to *bsum* (i.e., basic sum expression), since it is given by

$$\texttt{pol}(\texttt{abs}(\texttt{pol}(y,[-2,4])),[-2,0]) + \texttt{pol}(\texttt{abs}(\texttt{pol}(y,[3,3])),[4,2])$$

To solve equations that involve *sum expressions* one may need to know how to solve $X^n \pm Y^n = 0$, $\sqrt[n]{X} \pm \sqrt[n]{Y} = 0$, for $n \geq 2$, or $X/Y \pm Z/T = 0$, with $degree(XT) \leq 2$ and $degree(YZ) \leq 2$. We notice that, in general we would not be able to solve the first two if instead of 0 we had a non-null constant k.

In the grammar, some categories have names that are indexed by 1, 2 or 12, because they result from the *basic* category when we restrict the degree to be 1, 2, or any of these two. As for $vquot_{12k}$ and $quot_{12k}$ the idea is that the numerator and denominator have degrees 1, 2, or 0. To avoid defining more grammar rules, the abbreviate notations $pol_1(T)$, $ipol_2(T)$ and $ipol_1(T)$ were introduced. For instance, $ipol_2(\texttt{pow(x,}N\texttt{)})$ rewrites to $\texttt{pol(pow(x,}N\texttt{),}[a,b,c]\texttt{)}$ by applying the rule (scheme) for $ipol_2(T)$.

4 Generating Exercises in a CLP System

CLP languages are quite convenient to constrain the exercises by imposing constraints on some variables of the problems' generator. In this way, constraints are useful to control the difficulty and adequacy of the exercises for a certain curriculum, stage or user. In order to test these ideas, we have developed a prototype of such a generator in SICStus Prolog [20] using CLP(FD) [4]. In particular, we would like to see how quickly it runs, so that we defined a predicate `examples/5` that obtains one exercise of each type for some given specifications, through backtracking.

```
examples(File,Degree,RateMin,RateMax,X) :-
  tell(File),
  constrs(CountTypes,urestr_function), % user-defined constraints
  undefined(OpMax),  CountOps #>= 0, CountOps #=< OpMax,
  Rate in RateMin..RateMax,  indomain(Rate),
  nl, nl,  write('>>>> rate':Rate), nl,  nl,
  function(Type,Degree,Rate,CountTypes,CountOps), % finds a type
  expression(Type,X,Expr),  % finds an instance
  write(Type), nl, write(Expr), nl, nl,
  fail.
examples(_,_,_,_,_) :- told.
```

E.g., if we launch `examples(probs2,2,9,12,x)`, the system writes expressions in the variable x, of degree 2 and difficulty level in 9..12 to the file `probs2`.

The actual meaning of such difficulty rate may be settled by the user who may be given permission to assign a rate to each type (for details, see `user_rate/2` below). The overall rate of an exercise is then the sum of such rates. Different and more sophisticated criteria shall be investigated. The expressions of a given *degree* (in this example, we asked for expressions of degree 2) shall evaluate to polynomials of that degree when simplified to get rid of `abs` and `pow`, and do not contain quotients and radicals.

It is quite impressive how quickly the program may obtain a huge number of expressions. Throughout this section, it is assumed that the reader is familiar with CLP languages, and in particular CLP(FD) (for an introduction and some references, see e.g. [13]).

We note that the constraint solver shall be mainly used to do *consistency checking* and *to propagate constraints* on the exponents and on the number of occurrences of some combinations of particular function types. So, none of the optimization facilities of the CLP systems shall be utilized.

4.1 Defining Type Schemes for Expressions

In the implementation, a higher level of abstraction is considered to denote types of expressions, as given in the leftmost column of the table shown above. This is done by introducing *type schemes with constrained finite domain variables* to define sets of expressions of the same form, that is to represent *expression templates*. They almost mimic the grammar categories, and the main idea is that the composition of functions (herein denoted by `o`) is the main operation to enable the definition of complex functions from the elementary ones. Hence, for example, `pow(_) o ip(1) o (p1 o x/p1 o x)` represents `pow(`$ipol_1(pol_1($`x`$)/pol_1($`x`$)), N)$, which, by the grammar, is a *sumexpr*. The following expression is a particular instance of this type scheme

$$\left(-2\frac{-2x-1}{-3x+4}+3\right)^7$$

and has type `pow(7) o ip(1) o (p1 o x/p1 o x)`. Here, `ip(1)` and `p1` replace $ipol_1$ and pol_1, respectively. In general, the grammar rules are implemented by predicates of the form

category`(Type,Degree,Rate,CountTypes,CountOps)`

the main one, which has appeared previously in `examples/5`, is `function/5`.

```
function(Type,Degree,Rate,CountTypes,CountOps)
```

The parameters `Degree`, `Rate`, `CountTypes`, `CountOps` are used to constrain the resulting scheme `Type`. This allows to impose constraints to control the difficulty level or form of the generated expressions and to tackle user-defined constraints.

4.2 Illustrating the Generation of Type Schemes

In the implementation, we distinguished the basic constructs for the *basic* and *vxip* grammar categories as polynomial or non-polynomial functions.

```
npftype(abs).   npftype(rad(_)).   npftype(pow(_)).
pftype(p1).   pftype(p2).
```

They may be *composed* to produce more complex types. The functions to which a basic function may be applied (i.e., composed with) are defined by `ctype_/2`.

```
ctype_(T,x) :- pftype(T).
ctype_(p1,T o _) :- npftype(T).
ctype_(T,xip(_,_)) :- npftype(T).
ctype_(T,Tc o _) :- npftype(T), (pftype(Tc); (npftype(Tc), T \= Tc)).
```

We note that, for instance, the second clause avoids forms as `p1 o p2` and `p1 o p1`, which are not possible by the grammar, either. To provide some further intuition on the implementation, we give the code of a predicate that defines the grammar category *basic*.

```
basictype(xip(1,1),2,Rate,Ts,0) :-
   user_rate(p2,Rate),
   sum(Ts,#=,1), increment(p2,Ts).
basictype(p2 o pow(N) o x,Dgr,Rate,Ts,2) :-
   user_rate(bisqr(N),Rate),
   sum(Ts,#=,1), increment(bisqr(N),Ts),
   degree(bisqr(N),Dgr).
basictype(T o Tc,Dgr,Rate,Ts,Ops) :- (pftype(T); npftype(T)),
   degree(T,DgrT), DgrC #>= 1, Dgr #= DgrT*DgrC,
   rate_restr(T,Rate,[RateC]),
   types_restr(T,Ts,[TsC]),
   ctype_(T,Tc),
   ops_restr(Ops,1,[OpsC]),
   ((Tc = x, DgrC #= 1, OpsC #= 0, RateC #= 0, sum(TsC,#=,0));
     basictype(Tc,DgrC,RateC,TsC,OpsC)).
```

Here, `Rate` is a parameter that allows us to have some control on the application of each of the clauses that define a predicate. It must be either instantiated or have an upper bound when `function/5` is called. This is important also to guarantee that the generation terminates. User-defined rates are assigned through `user_rate/2` to the primitive functions and to particular sub-expressions (as for example, sums of radicals, quotients and products). The overall rate is then the sum of such rates, as we mentioned before. Since the teacher is not supposed to learn CLP to be able to constrain the generated problems, such a system shall have an user-friendly interface. For that purpose, we are investigating the use of the *Pillow* package [3] to develop a web interface. It is not straightforward to decide which kind of constraints we shall let the teacher define, since we would

like to achieve high flexibility and expressiveness, but keep the parametrization task simple.

Now, `Ts` is a list of finite domain variables, each one giving the number of occurrences of a given type. These types include `p1`, `p2`, `abs`, `rad(_)`, `pow(_)`, `xip(_,_)`, `bisqr(_)`, but also more general ones as, for instance, `prodstype`, `divstype` and `sum`. The latter is related to the expressions identified by *sumtype* in the grammar. The idea is that the user may define constraints on the values of the counters in `Ts`. These constraints may involve a single variable (e.g., to specify its domain) or any subset of them. Calls to `constrs/2` result in imposing the user-defined constraints on `Ts` for the category identified by `urestr_`*name*. The following example is meant as a mere illustration. It establishes that the number of `abs`, `bisqr(_)`, `pow(_)` and `rad(_)` shall not exceed four and that there shall be at least one `abs` and one `bisqr(_)`.

```
user_function(Ts) :-
   elements([abs,bisqr(_),pow(_),rad(_)],Ts,Vars),
   sum(Vars, #=<, 4).
   elements([abs,bisqr(_)],Ts,[Abs,BSqr]), Abs #>=1, BSqr #>= 1.
```

Predicates as `rate_restr/3` and `types_restr/3` (that appeared in the third clause of `basictype/5`) propagate the constraints for rates and type counters, respectively whereas `increment/2` tries to increment a given type counter by 1. The number of operations (i.e., compositions, sums, products and quotients) may be limited and `ops_restr/3` is used to propagate such constraints.

The use of a constraint language helps simplify the implementation. As an example, $basic_{12}$ is just `Dgr in 1..2, basictype(T,Dgr,Rate,Ts,Ops)`, if `basic/5` implements the grammar category *basic*. Nevertheless, it is still not easy to implement a constrain-and-generate strategy in order to avoid introducing what can be seen as symmetries in types. Indeed, being `+` a commutative operator, `abs o p1 o x + abs o p2 o x` and `abs o p2 o x + abs o p1 o x` should be the same type. Some symmetries may be filtered out by propagating constraints on the number of operators.

Finite domain variables could have been used to restrict the type constructs that are applicable at each derivation step. This improvement shall be analysed for future implementations.

4.3 Finding Particular Expressions

Instances of the expressions of a given `Type` may be obtained by calling

```
expression(Type,X,Expr)
```

For each type scheme, we may generate several expressions of that type by repeated calls to `expression/3`. Variations of the same example, in which the coefficients and exponents may change, can be easily found by forcing backtracking, in the CLP framework.

The predicate `function/5` generates a type scheme that may contain domain variables (representing exponents) with some attached constraints. Now, instead of saving all the constraint store on the exponents for later usage, the system would rather either save a particular instance of the type scheme or some pre-defined number of expressions that conform the type scheme. Different algorithms may be implemented to define `expression/3`, which may be even specialized to the particular problem we have in mind.

One possibility was described in Example 1, but we may also simply compute coefficients at random, though within a given range of pedagogical interest. Another possibility could be to use the program to generate several exercises which would later be filtered out, in view of the special application.

When only partial consistency is enforced, we have to guarantee that the (random) labeling process eventually stops, when no solution exists. The program currently implements committed-choice, disallowing backtracking to the random numbers generator when a feasible value is found to the to variable that is being labeled. In this way, the program may fail to find a solution even if one exists. This problem is not specific of CLP and other strategies could be devised to overcome it.

The type scheme plays a crucial role not only in the generation phase but also to render the implementation of problem solvers easier. We are mainly using CLP(FD) to generate the expressions, which then naturally would have integer coefficients. We have also made some simple experiments with other constraint programming domains, namely CLP(R), to define and tackle some conditions on the final expressions. However, the preliminary results had almost no interest for educational purpose. Further experiments could be done.

5 Symbolic Processing to Compute Roots and Domains

In general, the system needs to support symbolic processing of algebraic expressions to provide exact representations of the solutions. Indeed, CLP(Q) [8] could be used for finding the solutions, but expressions should have degree 1 and not involve the `abs` construct, so that they would be quite elementary.

We have already implemented a prototype program that finds the zeros and the domains of some of the expressions. For that purpose, we have implemented a symbolic solver for single constraints, that can handle any of the relational operators ($\geq, \leq, =, \neq, <, >$).

The arithmetic operations that involve only rationals are dispatched to the CLP(Q) solver. We are not using CLP(R) because we want to have exact (symbolic) representations for the irrational numbers that are handled, instead of floating point numbers. This is usually important for educational purposes.

Symbolic manipulation of irrational numbers is supported for the special forms $r_0 \sqrt[r]{r_1}$, $r_0 + r_1\sqrt{r_2}$ and $r_0 \sqrt[n]{r_1 + r_2\sqrt{r_3}}$, where the r_i's stand for rational numbers. For educational purposes these forms are already too sophisticated for the common intended users. We introduced some normal form $r_0 \sqrt[r]{r_1}$ so that the system would reduce, for instance, $\sqrt[3]{-40}$ to $-2\sqrt[3]{5}$, $\sqrt[6]{4}$ to $\sqrt[3]{2}$, $\sqrt{\frac{1}{2}}$ to $\frac{\sqrt{2}}{2}$,

$\sqrt[3]{\frac{1}{2}}$ to $\frac{\sqrt[3]{4}}{2}$. When higher exponents occur, the numbers may exponentially grow if we apply the latter transformation, so that we may possibly not keep it in future versions of the system. Prime factorizations of integers are found to make these transformations. Since we are not planning to tackle very large numbers, this apparently is causing no inefficiency.

To solve problems that require finding the domain of a function, the system needs to exactly solve disequations and disjunctions. Furthermore, we would like to be able to provide explanations of the solving steps. For both these reasons, the CLP(Q) solver, acting as a black-box, cannot be utilized to discard symbolic manipulation of constraints, even when no irrationals are involved. The same can be said about CLP(R), which in addition would yield floating point numbers.

We have not annotated the programs to explain the solving steps as we have done for the Maple programs. We think that the explanations that we obtain by introducing simple annotations in strategic points of the recursive solving procedures are often pedagogically poor. Because of that we shall investigate how to construct the explanation given all the solving steps.

6 Towards making the system available in the Web

We would like to obviate the need for students to learn a special syntax just for typing and reading formulas on the computer, unlike in *WebMathematica* [14], for example. Although textual representations for mathematical expressions are quite common, for example, in programming languages, they are not suitable for a learning environment (although, not everyone agrees with this [10]). The presentation and editing should be as close to the classical pencil-and-paper notation as possible.

To illustrate the potentials of the program, we have written some predicates to convert the internal representations of the mathematical expressions and solutions to LateX. This allowed us to pretty print the mathematical expressions.

Another possibility, that we considered at start and may further investigate, amounts to use a prototype viewer/editor of MathML documents, written to Tcl/Tk (some more details may be found in [23]). MathML is an instance of the XML markup languages for encoding documents containing mathematical formulas [15]. It is designed not only to present maths on-screen (i.e., on web documents) but also to support high-quality printing, rendering for non-graphical readers (for example: audio readers) and to serve as input/output language between math-aware software (for example: computer algebra systems).

Unfortunately, there are currently few web browsers that support MathML directly (one example: Amaya [1]), and none of the more common browsers support it "off-the-shelf". The situation is likely to improve in the future, either by the use of plug-in applications or updated browsers. To visualize the MathML document we used our own prototype viewer/editor, written using the Tcl/Tk toolkit [21]. This prototype viewer supports just a small subset of the *presentation* MathML, but this subset is all our MathML documents use. The quality of

the layout is comparable to that obtained using *Maple*, with the advantage that text can be freely mixed with math formulas.

By choosing a subset language from MathML we gained a good framework for further increasing the expressiveness of our presentation language should the need arise in the future. Furthermore, when MathML is directly supported by web browsers, we can easily integrate our documents in web pages.

In the mean time, further work in this viewer will include integrating it into web pages by making it a *Tcl plugin* (a small application that can be executed within standard web browsers).

One further advantage is that this viewer is also a *graphical editor* where math expressions can be entered and modified in two-dimensional layout (unlike in Maple, for example, where expressions are always edited in textual form). In the future, we intend to allow the student to interact with the tutoring system (for example, to input trial solutions for the exercises) and this will avoid the need to learn a special syntax.

7 Conclusions

We proposed a methodology for designing online exercises systems with special focus on applications to Mathematics education. The emphasis is on working backwards from the intended solution of the problem to obtain a sequence of steps leading to that solution. Prototype programs using CLP show that these languages have the right expressiveness to encode control on the system in an elegant way. The main drawback is that we cannot take complete advantage of CLP solvers to reduce the implementation effort. Indeed, we need to handle symbolic representations of some types of irrational numbers. Moreover, we also need symbolic processing of constraints, for example, to be able to find the domains of functions or to provide explanations. Since the system must have great control on the solving procedure to be able to explain the solving steps, we think we would not benefit if we used other languages and platforms to implement the system.

We shall analyse the integration of the system in a web-based environment. In particular we study the integration in Ganesh [11], although, so far, this distributed learning environment has been mainly used for Computer Science topics, with an emphasis on the automatic grading and correction of students exercises.

References

1. *Amaya, W3C's editor/browser*, W3 Consortium. `http://www.w3c.org/Amaya`
2. Bryc W., Pelikan S.: Online Exercises System, University of Cincinnati, 1996. (`http://math.uc.edu/onex/demo.html`)
3. Cabeza D., Hermenegildo M., Varma S.: The PiLLoW/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems. In *Proc. of the 1st*

Workshop on Logic Programming Tools for INTERNET Applications, JICSLP'96, Bonn, 1996. (http://www.clip.dia.fi.upm.es/miscdocs/pillow)
4. Carlsson M., Ottosson G., Carlson B.: An Open-Ended Finite Domain Constraint Solver. In *Proceedings of PLILP'97*, LNCS 1292, 191-206, Springer-Verlag, 1997.
5. Cohen A. M., Cuypers H., Sterk H.: *Algebra Interactive*, Springer-Verlag, 1999.
6. Gang X.: WIMS – An Interactive Mathematics Server, Journal of Online Mathematics and its Applications, 1, MAA, 2001. (http://wims.unice.fr)
7. Geometer Sketchpad, Key Curriculum Press. (http://www.keypress.com/sketchpad)
8. Holzbaur C.: OFAI clp(q,r) Manual, Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.
9. Kent, P.: Computer-Assisted Problem Posing in Undergraduate Mathematics, Institute of Education, University of London 1996. (http://metric.ma.ic.ac.uk)
10. Klai S., Kolokolnikov T., Van der Bergh, N.: Using Maple and the web to grade mathematics tests, International Workshop on Advanced Technologies, Palmerston North, New Zealand, December 2000 (http://allserv.rug.ac.be/~nvdbergh/aim/docs)
11. Leal J. P. and Moreira N.: Using matching for automatic assessment in computer science learning environments, *Proceedings of Web-based Learning Environments Conference*, 2000. (http://www.ncc.up.pt/~zp/ganesh)
12. Maple, Waterloo Maple Corporate. (http://www.maplesoft.com and http://www.maple4students.com)
13. Marriott K., and Stuckey P.: *Programming with Constraints – An Introduction*, The MIT Press, 1998.
14. Mathematica, Wolfram Research Inc. (http://www.wolfram.com/products/mathematica)
15. *Mathematical Markup Language (MathML) Version 2.0*, W3C Recomendation, February 2001. http://http://www.w3c.org/TR/2001/REC-MathML2-2001021
16. Melis E. et al.: ActiveMath: A Generic and Adaptive Web-Based Learning Environment, *Artificial Intelligence in Education*, 12(4), 2001. (http://www.mathweb.org/activemath)
17. Moore L., Smith D. et al.: Connected Curriculum Project CCP, Duke University, 2001. (http://www.math.duke.edu/education/ccp)
18. Sangwin C.J.: New opportunities for encouraging higher level mathematical learning by creative use of emerging computer aided assessment. University of Birmingham, UK, May 2002. (www.mat.bham.ac.uk/C.J.Sangwin/)
19. Schrönert M. et al.:*GAP – Groups, Algorithms, and Programming.* Lehrstuhl D für Mathematik, Rheinisch Westfälische Tecnhische Hochschule, Aachen, Germany, 1995.
20. SICStus Prolog User Manual Release 3.8.6, SICS, Sweden, 2001. (http://www.sics.se/isl/sicstus.html)
21. The Tcl developer Xchange, Scriptics. http://http://www.scriptics.com
22. WeBWorK, University of Rochester, 2001. (http://webwork.math.rochester.edu)
23. Tomás A. P., Vasconcelos P.: Generating Mathematics Exercises by Computer. Internal Report DCC-2001-6, DCC - FC & LIACC, University of Porto, 2001. (presented at workshop CSOR'01, Porto, 12/2001)