Parametric exercises via Zope server

Jordi Saludes*, Sebastià Martín** and Miquel Dalmau*

28th June 2002

* Dpt. ma2, Universitat Politècnica de Catalunya, Edifici TR5, Colom 11, 08222 Terrassa. *e-mail*: {saludes,mdalmau}@ma2.upc.es

** Dpt ma4, Universitat Politècnica de Catalunya , Campus Nord, c/Jordi Girona, 1-3, 08034 Barcelona. *e-mail:* sebasm@mat.upc.es

Abstract

Symbolic mathematical software is now becoming widespread in mathematical education. This paper describes a new way of using such tools in a web-based learning environment providing mathematical exercises with parameters depending on the identity of the student. Namely, we present a way to use a symbolic mathematics package or library for displaying, correcting and grading template exercises with mathematical content. We describe also the planned transition from a hardwired *cgi* system to a much more flexible system using open-source code.

1 Introduction

The usual setting for a traditional math lab class starts when the teacher hands off the exercise sheets. The students are supposed to provide answers to the questions therein by working out the exercises using the computer as an enhanced calculator.

In these practical sessions, the teacher role is twofold: Assisting the students in the solving process and assessing the validity of the answers. If the answer is wrong the teacher explains why it is not acceptable and the student returns to work.

Nowadays, it is clear that we can charge the mostly automatic and unsympathetic role of assessment to the computer and leave the more sophisticated and friendly task of helping the students to the teacher. This setting becomes the only way to assist the students when the ratio of students per teacher and the amount of exercises are high.

1.1 Description

We will call *parametric exercise* a problem, involving mathematics, in which the actual data presented to the student depend on external parameters. For instance, we can use the student's identity as a parameter to generate instances of a problem (of similar difficulty) with different data in order to avoid the possible copy of answers among students; or use a parameter to control the difficulty of the generated exercise (where this parameter is deduced from the evolution of the student).

In this paper we describe an initial design to use a symbolic mathematics package or library for displaying, correcting and grading parametric exercises using mainly open-source software components. The system essentially consists of:

- A web server for user interface which works as a backbone for the different components;

- A symbolic mathematical package (the *mathematical engine*) for generating mathematics objects acting as parameters in the exercises, and for checking the student's answers.

In section 2, we describe how the system works at present in an Engineering School. We have been using the system for two years with a very low rate of drop-out. We think it is now the time for extending the experience to other courses and teachers. In fact we plan to develop the system as open-source to target a large community of university and secondary school teachers which —hopefully— would adopt the system for developing their own exercises.

On the first hand, to have a usable tool for a large community of authors we need an installation procedure as straightforward as possible; but on the other hand, the system has to be flexible enough to satisfy a diversity of tastes and allow a fair amount of customization. Moreover, we need flexibility to use the system as a test-bed to explore different rendering schemes and mathematical packages. Therefore, high in the requirement list is the need to build the system in a complete modular form and mostly independent of the particular mathematical engine being used. In section 3 we explain how the system will be organized and we introduce two examples to show how the student/computer interaction will go.

The brain of the system resides in the mathematical engine: A server application providing all sorts of mathematical computations to the user front end. In section 4 we sketch the programming interface to such a server and we discuss two alternatives for implementing such engine.

2 The system at present

Our system is used nowadays in a course (Mathematical Methods 1) with about 200 undergraduate students. The students use the system as an exercise repository, providing both correction and reckoning of exercise solving status.

Based on a hard-wired `cgi` connecting to a *Mathematica* kernel [2], it delivers mathematical objects of the following types: polynomial, rational in binary radix, integer and float and lists of these types, in a format readable to a user unaware of the *Mathematica* syntax.

Conversely, it parses user's answers into objects of the above types. These objects are then checked versus a suite of mathematical tests to assess the correctness of the user's answer.

2.1 Example

Let us see a test suite for checking whether an object $p(x)$ of type polynomial interpolates points $\{(x_1, y_1), \dots, (x_n, y_n)\}$

1. Check whether the student input string can be parsed as a *polynomial*.
2. Check whether the degree is the right one.
3. For each interpolating point (x_i, y_i) $i = 1, \dots, n$ check whether $p(x_i) = y_i$.

If any of the previous tests fails, appropriate feedback is provided by the system to the student.

3 The system in the future

We plan to migrate the system from a monolithic *cgi* program to a software component which could be added to some web servers (see section 3.2), in such a way that new exer-

cises could be developed by users (the teachers) with only a mild knowledge of the mathematical engine which lies in the backstage. Another goal is to provide independence of the kind of mathematical engine being used.

3.1 Kinds of users

Three kinds of users need to deal with parametric exercises:

Students People who use the exercise.

Author The person writing the exercise, setting the variables and the associated documents.

Manager The user who creates and/or installs new variable types.

3.2 The web server

The first component is a Zope [1] server/database to which the student will connect. The server task consists of storing all the material the exercise is composed of, and, on the other hand, delivering this material on demand.

Zope is an open-source web content management system with a very flexible system of permission/roles based in the Python language [3]. In this system all the documents are organized in folders like a directory tree.

This allows the web-master to delegate responsibility for documents to the owners of folders (directories). Another fine point —and essential for our needs— is that its documents may be written in DTML, a superset of HTML marking set, providing:

- Displaying of variables through in an HTML document.
- Iterating on sequences.
- Conditional display of parts of the document.

Zope also supports other types of logical marking in documents compatible with these properties: Using XML namespaces in the so-called *Template pages* or full-blown XML/XSL documents. These features are added to the Zope server by installing a suitable *product*:A software component in the form of one or several Python classes. Although XML document types are more complete and standard, they are also more difficult for authors to learn and manage; so we initially stick to DTML document as the main document type.

Zope is extensible by way of the *products* mentioned above: Python classes with user interface given by a set of DTML methods/documents. We plan to use one of these products to glue the web management part with the mathematical core part.

3.3 Objects

The solving of a mathematical exercise sometimes resembles the sequence of procedure calls in an imperative computer language: The goal of an exercise is making the student produce a mathematical object fulfilling a set of conditions. This goal may be achieved by following one of several strategies, which, in turn, are decomposed in a sequence of sub-goals.

A mathematical learning environment should address not only the displaying of the exercise and the correction of student answers: When the student is not able to get the exercise done by himself, the system has to provide contextual help on demand about the solving strategies. Thus, it is convenient that contextual help contain links to other parametric exercises devoted to the resolution of the sub-goals of the main exercise; and that

Table 1: Relations between a procedure and a *PEFolder*

PE environment	Imperative language
PEFolder	procedure
PEVar	local variable
parameters in a GET or POST http request	input arguments
`if` tags in DTML documents	Bifurcations

these exercises be instantiated with the actual data the student is working with, much in the same way a procedure calls another with given input parameters.

For this reason we have modeled our *parametric exercise* class like a procedure, with the following elements:

PEFolder A folder-like container which has *PEVar*s as properties and the rest of elements as sub-objects.

PEVars They work like two-faced variables: A face from the user to the mathematical engine, with a `parse` method for converting user strings to mathematical values. Another face towards the student, with a `str` method for displaying the variable and a `form` method to show an input box to be filled in by the user.
A switch determines if the *PEVar* is used either for collecting user's answers (*input*) or for displaying parameters (*output*).

Script A `init` script is executed when starting a session. This is the moment to set all *PEVars* to a fixed value or to a random one. Another script will check the user's answers and will prepare *PEVars* for the next step.

DTML Documents providing the user's interface for displaying the exercise text, give feedback and collect the user's answers.

PEVars are the central objects of the setting. They connect the user external representation of a mathematical object to the inner representation as an expression in the language of the mathematical engine.

3.4 Types on *PEVars*

Variables on a *parametric exercise* should be typed for several reasons: We can think of a *PEVar* as a mathematical variable enriched with user's interface methods, specially `parse`, `str` and `form`. Input string representations are converted in mathematical engine expressions by parsing, and this representation is specific to the type. The same consideration applies to the output representation given by `str`.

Moreover, even at the mathematical level, we need the same operator to behave differently from one type to another. For instance, it is not the same to test for equality on rationals as in floating point numbers. In the first case we want an exact match, in the second case it is more useful to test the relative error.

3.4.1 Base types

We will define base classes **Numeric** and **Ordered** supporting basic arithmetic (`+`,`-`,`*`,`/`) and the boolean operators (`<`,`>`,`==`) to be used by the teacher (see section 3.1) when writing the exercise documents. Other types will be derived from these base types.

3.4.2 Compound types

New types can be composed from others like **List P** which stands for an ordered list of objects of type **P**, or **Set P** for an unordered set of elements of type **P**. Some of the operators for a compound type are automatically deduced from the base type. For instance, the evaluation of the equality operator for **List P** reduces to the logical intersection of the evaluation of the equality operator of type **P** for corresponding elements in the list.

3.5 Two more examples

We now present two examples using types derived from type **Rational**. One related to the binary representation (**BinaryPointRational**) and another to a primary school example, related to the simplification of fractions (**NonReducedRational**)

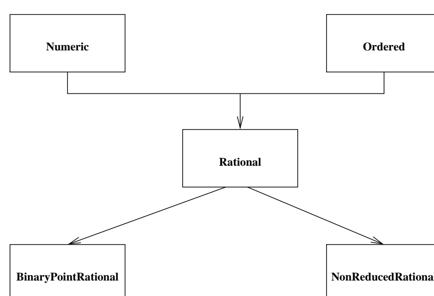

Figure 1: Hierarchy of types used in the examples

3.5.1 A numerical example

One of the exercises we propose in the course "Mathematical Methods 1" deals with the floating point representation of real numbers. Let us suppose, for instance, that we want to write a *parametric exercise* about converting a rational number to binary form. The teacher has to write this exercise in a *PEFolder* having two *PEVars:*

- The number r presented to the student for conversion, of type **Rational**.
- The variable s for collecting the student's answer, of type **BinaryPointRational**.

The second type is a specialization of **Rational** with a different parser to accommodate binary representation of rationals. Let us show how a student's session will work:

1. The student connects to the URL corresponding to the *PEFolder* of the exercise. This makes the mathematical engine to set the random seed based on the student identity and to evaluate the `init` script. In this script we set all the parameters we need for the exercise to be displayed and checked. In this case, the *PEVar* r is randomly set to, say, $15/7$.
2. Then, the `index_html` document is rendered for the student. It contains DTML tags that display r in the form `15/7' and $s$ as a HTML `input` element for the student to fill in. We expect the student to convert $15/7$ into the form

   $$10.00\%100$$

   where the part after the percent sign corresponds to the periodic part in the binary representation of the number (that is $10.00100100100\ldots$).

3. The student writes his/her answer in the corresponding form and clicks on the `Check' button.

4. Now the system pipes the student's string `10.00%100' to the parser for type **BinaryPointRational** which converts it to a math engine expression (like `Divide[15,7]` in *Mathematica* language). In case the parser fails, this condition is signaled to the user as "Can not read your input as a binary point rational number" . Otherwise the attribute `value` of s is set to this expression.

5. A script is executed to check if the value of s is right. This may be a DTML Method, containing expressions to be evaluated by the mathematical engine in order to assess the correctness of the student's answer, with a chain of `if` commands to dispose and give feedback for each case of possible error.

6. Changes in the values of variables are kept in the engine process associated with the connected student.

7. A right answer may imply calling a method for setting a mark in the student database, in order to track the performance of the student.

3.5.2 A primary school example

In this example we discuss an exercise for helping the students grasp the simplification of rational numbers.

Here we will use the type **NonReducedRational** which is also a type derived from **Rational**. Since we will need to deal with student's inputs representing a non-reduced rational, it is clear that the parsing method for type **NonReducedRational** can not pass a fully simplified rational to the mathematical engine. We need instead a semi-evaluated form. When the system has to do rational arithmetic, this semi-evaluated form will be reduced in full.

For example, if the student submits the string '4/6' and we use *Mathematica* as a mathematical engine, the two reduction steps will be:

`'4/6'` $\mapsto$ `NonReducedRational[Divide[4,6]]` $\mapsto$ `Rational[2,3]` ;

where `NonReducedRational` is an inert function used to identify the type of the object. Note also that this function must have the attribute `HoldAll` set in order to avoid the automatic simplification of $4/6$ to $2/3$.

On the other hand, two new methods `(numerator` and `denominator)` are defined to extract the two components of a fraction. With this new type the exercise will proceed as follows:

1. Two *PEVars* p (output) and q (input) of type **NonReducedRational** are defined. At the beginning the system sets p using the random initializer and it assures p is not reduced.

2. The system displays p and ask for the simplification to be put in q.

3. The student value in q is parsed, then the system computes g: the greatest common divisor of the numerator and denominator of q.

   (a) If $g = 1$, the system congratulates the student and finishes.

   (b) Otherwise, it moves q to p and informs the student that the exercise is not yet finished. Back to step 2.

4 Mathematical engines

As we have mentioned above, all the mathematically relevant computations are done by the *mathematical engine*: A server application that silently listens for connections and provides answers to mathematical queries originated in the exercise documents or scripts. We plan to test two implementations of the mathematical engine using a common application interface which will be modeled after the *MathLink* API [2, 4]. Supported functions will include:

Initialize: Begins a connection with the mathematical engine by providing hostname and port. It returns a structure used to connect to this particular process. This function must be called when a new student logs in the system: it starts a new process in the server end, which will hold *PEVars* values for the new user and will last until the student logs off or a given time interval of inactivity has passed.

Open, Close: The first function is used to connect to a designated running process. The second function is used to cleanly close the connection while keeping the remote process listening for a new connection.

Finish: Disconnects from the mathematical engine process and terminates it.

SetContext: Switches to the context denoted by a string argument. Contexts works as namespaces allowing *PEVars* with the same names coexist in the same process.

Source: Loads the definitions in the file designated by a string argument or read them from a stream argument. It is used to pass native language expressions to the server.

SetVar: Creates a new mathematical value in the remote process associated with a *PEVar* by name. The parameters passed on call are: Name of the *PEVar*, string form of the *PEVar* value and type.

PutFunction: Pushes the function named in a string argument into the engine evaluation stack. A second integer argument specifies the function arity.

PutVar: Pushes the named *PEVar* value into the evaluation stack.

GetBool, GetString, GetInt Gets a boolean result from the evaluation stack as an integer ($0 =$false, $1 =$true). The other two functions do the same for strings and *C*-style long integers.

4.1 The *Mathematica* way

The first possible implementation of the mathematical engine is based on the current implementation of the system based on *Mathematica* [2]. It is a full-featured symbolic mathematics package in a server-like fashion (the kernel) along with a C library for connecting [4] with it. Required adaptation is minimal in this case, consisting only in wrapping *MathLink* calls to form the engine API. Moreover, the concept of *context* is already present in the *Mathematica* language.

In this case *PEVar* values are represented by *Mathematica* language variables and the type methods (`set`, `parse`, `str`) will be described in terms of constructor functions like `NonReducedRational[4/6]` in the example of section 3.5.2. We will use *Mathematica* pattern matching mechanism to write the type-specific methods. For example, the definition of the `equal` method for this type (used to decide whether two **NonReducedRational**s are the same) will begin

```
equal[NonReducedRational[a_],NonReducedRational[b_]]
:=...
```

4.2 The *Haskell* way

A compiler for the *Haskell* language [5] is a good candidate for an alternate implementation of the mathematical engine. *Haskell* is a functional language, statically typed, that seems to fit naturally in the proposed scheme for mathematical object types. The language has base classes for **Numeric** and **Ordered** already defined (in the `Prelude` module). It allows also the combination of existing types to form compound types (see 3.4.1 and 3.4.2).

Moreover, programs in functional languages are more suitable to formal proof of correctness[6], which might be relevant when designing parametric exercises. Namely, proving that all instances of a parametric exercise have a similar difficulty, or that a test for a student answer will always give the appropriate result.

Another benefit from using this approach is that there is a *LALR* parser generator for the *Haskell* language [7] similar to the *yacc/lex* suite for the *C* language. We do not know of a similar tool for the *Mathematica* language.

On the negative side, we have to point out that *Haskell* is not a language dedicated to higher mathematics, but we have to take into account that the main goal of the system is to check whether students solutions are right and usually this task is far easier than actually solving the problem (which is the point the mathematical packages like *Mathematica* are good for).

And last, but not least, there are a couple of open-source compilers for *Haskell.* Using one of them as the foundation of the mathematical engine will permit to release the system as public domain software.

5 Conclusions

We have presented an experience on using a mathematical package for checking and giving feedback to students. A design for developing the original system into a modular system joining a standard web-management system and a mathematical sever has been presented. We considered the points of view of the different users: The student and the author, and we propose a place-holder (the *PEVar*) as a link joining the representation of a mathematical object at the user level with the internal value lying in a mathematical engine process. We have discussed the concept of *PEVar* type as an organizing principle for the entire system and presented two possible implementations for the mathematical engine.

References

[1] `http://www.zope.org`

[2] `http://www.wolfram.com`

[3] `http://www.python.org`

[4] S. WOLFRAM. *The Mathematica Book*. Wolfram Research (1999)

[5] `http://www.haskell.org`

[6] JOHN HUGHES. Why Functional Programming Matters. In D. TURNER, ed. *Research Topics in Functional Programming*. Addison-Wesley (1990)

[7] `http://www.haskell.org/happy`