

Using camlp4 for presenting dynamic mathematics on the web: DynaMoW, an OCaml language extension for the run-time generation of mathematical contents and their presentation on the web

Frédéric Chyzak, Alexis Darrasse

▶ **To cite this version:**

Frédéric Chyzak, Alexis Darrasse. Using camlp4 for presenting dynamic mathematics on the web: DynaMoW, an OCaml language extension for the run-time generation of mathematical contents and their presentation on the web. ICFP 2011 - 16th ACM SIGPLAN International Conference on Functional Programming, Sep 2011, Tokyo, Japan. pp.259-265, 10.1145/2034773.2034809 . hal-00640584

HAL Id: hal-00640584
https://hal.science/hal-00640584

Submitted on 31 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using Camlp4 for Presenting Dynamic Mathematics on the Web: DynaMoW, an OCaml Language Extension for the Run-Time Generation of Mathematical Contents and their Presentation on the Web

An experience report

Frédéric Chyzak *
INRIA (Rocquencourt, France)
frederic.chyzak@inria.fr

Alexis Darrasse *
LIP6 (Paris, France)
alexis.darrasse@lip6.fr

Abstract

We report on the design and implementation of a programming tool, DynaMoW, to control interactive and incremental mathematical calculations to be presented on the web. This tool is implemented as a language extension of OCaml using Camlp4. Fragments of mathematical code written for a computer-algebra system as well as fragments of mathematical web documents are embedded directly and naturally inside OCaml code. A DynaMoW-based application is made of independent web services, whose parameter types are checked by the OCaml extension. The approach is illustrated by two implementations of online mathematical encyclopedias on top of DynaMoW.

Categories and Subject Descriptors D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; H.3.5 [*INFORMATION STORAGE AND RETRIEVAL*]: On-line Information Services; I.1.3 [*SYMBOLIC AND ALGEBRAIC MANIPULATION*]: Languages and Systems

General Terms Languages, Web Applications, Computer Algebra, Symbolic Computation

Keywords mathematical encyclopedias, quotations, antiquotations, metaprogramming, web services

1. Introduction

The communication of scientific knowledge is still commonly carried out via written media. Such documents are essentially fixed, even though they can now and then be brought up to date through successive editions. In contrast, the web has in recent times allowed a rapid and reactive diffusion of information, as well as an interactive access to electronic content. Encyclopedias and handbooks specialised in specific fields of mathematics are subject to this change from written media to web content, as their readers now prefer obtaining the mathematical formulas they need online, rather than from books on their shelves. Two ambitious realisations of this kind are the On-Line Encyclopedia of Integer Sequences[1], which is essentially a database with predefined mathematical queries, and the Digital Library of Mathematical Functions[2], which is a human-written mathematical text, specially typeset for the web. Both are successors of best-seller books [1, 5].

But other encyclopedias make essential use of symbolic calculations performed by computer-algebra systems, that is, calculations with terms representing exact algebraic quantities constructed over integers and symbols: polynomials, matrices, differential equations, etc. Two such projects are the Encyclopedia of Combinatorial Structures ("ECS") and the Encyclopedia of Special Functions ("ESF")[3]. There, the symbolic calculations were performed once and their results stored statically, before publishing the web sites. But, in the context of symbolically-generated mathematical knowledge, a natural desire is to perform symbolic computations at run-time. For instance, ECS identifies a combinatorial structure selected by the user through its first terms. The reply by the system consists in a collection of data statically attached to the sequence and merely retrieved from a database: name, combinatorial interpretation, references, etc. A natural wish of the user would be, via new requests, to obtain an arbitrary number of terms of the sequence, or to draw an instance of the structure randomly. This new, dynamic information would be calculated at the time of the request by a computer-algebra server. For its part, ESF provides, for a special function given by the user, a large number of formulas satisfied by the function, and several evaluation graphics. In this output, a series expansion of a certain order, for example, should be extendible at will, based on a user-supplied parameter. Similarly, graphics should be able

* Frédéric Chyzak was partially supported by the Microsoft Research – INRIA Joint Centre and most of the work was done when Alexis Darrasse was a postdoctoral fellow there.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'11, September 19–21, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0865-6/11/09. . . $10.00

[1] `http://oeis.org/`

[2] `http://dlmf.nist.gov/`

[3] The first author has maintained some part of ECS and has seen the development of ESF by close colleagues at INRIA.

to be recomputed and replotted interactively with respect to different ranges of the variable.

Part of our project consists in developing these encyclopedias towards more interactivity, and this triggered the development of the present work: a tool to help developing the above-mentioned encyclopedias. More than just assisting the typing and typesetting of mathematical formulas, a human-process that is prone to errors, what we needed was a programming environment for the dynamic (run-time) generation of (mathematical) symbolic content and its presentation online. We view this as *dynamic mathematics on the web*, whence the name of our library and system, DynaMoW[4]. Beside generated encyclopedias, we believe this tool can have applications to design environments whose users are no expert of computer algebra, but need some sort of symbolic calculations following a number of fixed scenarios. This would be the case, for instance, in education.

1.1 Desired Features

On the application level. Our main need was *interactivity*, requiring the ability for run-time symbolic (re)computations. This is well exemplified by our successor of ESF, the Dynamic Dictionary of Mathematical Functions (DDMF)[5] [3]. For example, its end users need to change parameters like: ranges of plots, orders of asymptotic expansions, precisions in numerical evaluations, etc. Likewise, interaction with ECS allows randomness in a page, to display random generated combinatorial structures. Thus, the system to be developed as a programming environment had to be able to simultaneously monitor calculations in a computer-algebra system and the generation of mathematical web documents.

System architecture: proper role of the computer-algebra system. ECS and ESF are based on the general-purpose computer-algebra system Maple[6] for their algebraic manipulations, but new encyclopedias could use other systems, including smaller ones specialised to various branches of mathematics. Besides, specific scientific communities with different habits could conceivably prefer another system, whence the need for possibly several cooperating computer-algebra systems in a future application.

For system tasks, both ECS and ESF have been excessively Maple-centric, with a minimal part of the software in Perl (the CGI). The choice of ESF to perform file manipulations and generation of XML documents inside Maple hinders the versatility towards various computer-algebra systems. Generating Maple and HTML by Perl (ECS) or just HTML by Maple (ESF) goes with heavy quotations of strings of the guest language in print formats, as computer-algebra system is typically not well suited for string manipulations. By not dissociating the application from this generation (ECS), any modification of the application requires changing the quotations again.

For both reasons, implementing the system in a computer-algebra system was not an option, and the main language naturally had to be a general-purpose language.

In addition, as we planned developing applications for end users with limited knowledge of computer-algebra, we could not expect any computer-algebra system to be installed on the user side. Thus, we confined ourselves to a client-server architecture, with no local computation.

Ease of programming and readability. The expected close interaction between symbolic calculations and the rest of the flow control suggested that computer-algebra code could not be kept separate from the rest of the source. In earlier projects, we had implemented as much as possible in the computer-algebra system, with induced heaviness. Here, we wanted several natures of programming languages to be mixed naturally in the same source files: a general-purpose language for the general control flow of the application; one or several computer-algebra languages; some string representation of literal fragments of web documents. Indeed, the ease to merge variables and HTML in PHP[7] has influenced our development.

Also, values have to migrate naturally between computer-algebra systems, control, and presentation, and, for readability, we wanted to be able to write a symbolic expression where it is used in the computer-algebra source code, rather than postponed (if using a notation of formats like with `printf`) or with an interruption in the syntax (if using a notation of concatenated strings). This all spoke in favour of a mechanism of *quotations and antiquotations*.

As to the general-purpose language, the wish for the comfort of a reasonably typed language immediately precluded a computer-algebra system as the main language. (One possible exception could have been Sage, which, however, was not so much developed at the time we started.)

1.2 Related Work

Ocsigen [2] is a complete system that provides facilities for programming the web in a type-safe environment. It is meant as a replacement for the pair Apache/PHP. It is written in OCaml and proposes an OCaml extension, Eliom, to develop OCaml applications. Our system DynaMoW works on the same level as Eliom and could be extended into a module for Ocsigen, so as to rely on it as a web server.

The free open-source mathematics software named Sage[8]. This project provides a unified interface to the major free computer-algebra systems in a minimal variant of the Python interpreter. Over the years, Sage has also become a computer-algebra system per se: besides libraries exposed from other systems, new code is being developed natively for Sage. It provides users with a textual read-eval-print loop, and thus is a system potentially used by DynaMoW[9]. Sage also exports a web-presented symbolic shell, which is thus mostly targeted at computer-algebra expert users. Sage embeds literally dozens of computer-algebra systems (whether big, general-purpose ones or smaller, dedicated ones). This makes it a large application (around half a gigabyte to download for the current release, 2.4 GB once uncompressed and installed). By contrast, DynaMoW is designed with web presentation as its main application and is very lightweight. Another difference is that Sage implements all mathematical notions as objects, and manipulates them mostly by interpreted Python libraries, with critical sections written in the related but different compiled language Cython[10], while DynaMoW is intended for compiled OCaml applications and does not require programmers to use any OCaml object facility.

[4] `http://ddmf.msr-inria.inria.fr/DynaMoW/`. Roughly 3800 lines of OCaml code in the current version, DynaMoW-1.0.0beta.

[5] `http://ddmf.msr-inria.inria.fr/`

[6] commercial product by Maplesoft, `http://www.maplesoft.com/`

[7] `http://www.php.net/`. PHP: Hypertext Preprocessor.

[8] `http://www.sagemath.org/`. Sage Mathematics Software.

[9] Indeed, we have a prototype plugin to glue to Sage.

[10] `http://www.cython.org/`. Cython: C-Extensions for Python.

2. Interfacing a Computer-Algebra System

2.1 Types for Values in a Mathematical Shell

Few developers use computer-algebra libraries, and all general-purpose computer-algebra systems behave as symbolic shells, that is, users interact with them through a textual read-eval-print loop. This makes a notion of a *symbolic session* central to the application. The OCaml level in DynaMoW thus makes successive interactions with the symbolic session, and needs to refer to values that persist in the symbolic memory inbetween.

To allow for several computer-algebra systems to be used in the same application, parametrised types have been used to represent symbolic values: the type for external symbolic values is `'t DynaMoW.symb`, with an abstract type parameter `'t` exported by the module implementing a computer-algebra system. This parameter can itself be parametrised, to express different datatypes in the same computer-algebra system. Thus, a symbolic value is typically of a type `'a MyCAS.t DynaMoW.symb`: the first level of parametrisation protects against using a value from some computer-algebra system with another, while the second level can be used to protect against mixing types within the same system, provided the application properly defines and uses several types for `'a`.

2.2 Computer-Algebra Programs as Quotations

In order to embed computer-algebra programs as directly as possible inside OCaml source, the technique of choice is *quotations*, which are available in OCaml by using Camlp4, in the notation `<:id<...>>`, where `id` stands for an identifier of the kind of quotation, and `...` is the quoted text. We believe that quotations were introduced into the (O)Caml family by Mauny and de Rauglaudre [4].

The main kind of symbolic quotation is of the form `<:symb<...>>`, and quotes a literal fragment of symbolic code. The interpretation of such a quotation is, after evaluation of the code by a computer-algebra system, a reference to an externally-living symbolic value, represented by some type `'a DynaMoW.symb`.

The previous quotation is meaningful for values that can have no OCaml counterpart, and thus have to remain external. Beside this, we made the informal assumption that a computer-algebra system has types that correspond to OCaml `int`, `bool`, and `string`, and designed additional quotations of the forms `<:int<...>>`, `<:bool<...>>`, and `<:string<...>>`, to denote computations whose results can be returned as native OCaml datatypes. As typical programming in a computer-algebra system uses a lot of side effects, we added a quotation `<:unit<...>>` that evaluates some code in the computer-algebra system, without actually returning any value to OCaml.

Symbolic calculations need to be parametrised by OCaml-generated values. *Antiquotations* are the dual of quotations to address this need, and we followed the Camlp4 notation: the syntax `$(tau:expr)` appearing in a quoted fragment stands for an escape, to be replaced before symbolic evaluation with a representation of the OCaml expression `expr` of type `tau` that can be parsed by the computer-algebra system. Thus,

```
let a = 3 in <:symb< f($(int:a)) >>
```

is intended to evaluate `f(3)` in the guest language, typically a function call.

Each antiquotation is intended to take the role of a syntactic entity in the guest language. For example,

```
let a = 3 in <:symb< f$(int:a) >>
```

expands as `f (3)` rather than `f3`. When Maple is the guest language, the former results in a function call, while the latter would be a Maple name[11]. We do not go beyond this in ensuring the validity of the generated fragment, and in particular we perform no parsing inside quotations.

The type of a `<:symb<...>>` quotation is `'a MyCAS.t DynaMoW.symb`: if the application programmer wants to use different types of symbolic objects, it is his responsibility to confine the use of `<:symb<...>>` quotations to an unsafe section of the code, and to export safe functions, with more limited types for `'a`, to be used in the rest of the application.

2.3 Computer-Algebra System Plugins

Computer-algebra systems can be used with DynaMoW through dedicated plugins, which are modules that abide by a fixed interface. Currently distributed plugins are a well tested Maple plugin, a simple OCaml plugin that serves as an example, as well as experimental Mathematica and Sage plugins. User-implemented plugins can also be developed.

A module is considered a plugin when it exports what enables DynaMoW to view the computer-algebra system as a shell, interacting with it through a read-eval-print loop:

- a type `cas_code` to store the symbolic commands after expansion of antiquotations inside `<:symb<...>>` quotations;
- "evaluators" that run a symbolic command and return either a reference to the symbolic result (`evaluator_symbolic : cas_code -> 'a t DynaMoW.symb`), or a LaTeX representation of the result (`evaluator_latex: cas_code -> latex`), or an OCaml value (`evaluator_int: cas_code -> int` for an integer value, and similarly for boolean, strings, etc.);
- a serialisation function, used internally for argument passing between services;
- a function to return a string containing a reference to a symbolic value living in the computer-algebra system, used internally to implement the `$(symb:...)` antiquotation;
- a start and a reset function, used automatically by DynaMoW to monitor a separate session for each service, so that users do not need to run/quit sessions explicitly;
- additional pretty-printing and related functions.

To simplify the creation of plugins, the provided functor `DynaMoW_cas.Shell.Generic` should be well adapted to generate plugins for most computer-algebra systems with a command-line interface. This functor takes as input a brief module that contains a string for each symbolic command needed by DynaMoW.

2.4 Interaction between Functional and Imperative Programming Styles

DDMF is the main application developed using DynaMoW. It consists of a mixture of functional code in OCaml and of imperative code in Maple. As of version 1.6, it is made of:

- 5 kloc of OCaml with Maple quotations,
- 8 kloc of generated OCaml with Maple quotations,

[11] For the sake of comparison, the quoted text in the preceding example really expands in the implementation as `f( (3) )`, which has the same interpretation as the announced `f(3)`.

- 12 kloc of external Maple library (only a part is used).

At one end of the system, OCaml is used to organise the structure of the produced web documents, the interplay between the web services that produce them, and the interactivity with the user. At the other end, Maple libraries dedicated to the manipulation of the so-called "special functions" of mathematics are called by Maple quotations in the OCaml source. Additionally, one of our goals with DDMF was to keep traces of the underlying symbolic calculations, so as to be able to generate human-readable texts describing the ongoing calculations. Doing so was made possible by *instrumenting* some part of our existing Maple library, Algolib[12]. In quite a few cases, this instrumentation in fact required us to port some of the Maple code to OCaml. What we originally thought would be just syntactic transformation became in a few cases a more involved manual operation.

Maple originally had no kind of records or data structure with named fields. Although records have been introduced in the language (rather recently), efficiency issues remain and, more importantly, a huge amount of legacy code, including a large part of ours, does not use them. Instead, the traditional programming style in Maple bases on "lists", which are in fact immutable arrays of heterogeneous values. The same approach of dereferencing by the position applies to accessing the "fields" of most Maple data structures: terms in a polynomial, coefficients in a series, etc. Additionally, the design of a data structure in a Maple application makes very often the convention that a few of the first or last elements provide information on the rest, which is then a collection of values of the same type. Maple lists being immutable, they are copied when modified, which makes one avoid accumulating values in a list: this would cause a use of memory quadratic in the length of the final list and lead to a slowdown, owing to the interaction of the garbage collector. Instead, the typical idiom is to generate data into a mutable structure (a hash table), before flattening it into a list.

As OCaml lists behave differently, it made no sense to keep this heavier approach in the port, and we constructed OCaml lists directly, by mapping over an integer interval.

Finally, there remains needs for alternative iteration on an iterable Maple data structure and its OCaml counterpart. To this end, we provided two functions

```
symb_of_symb_list : 'a maple list -> 'a maple
symb_list_of_symb : 'a maple -> 'a maple list
```

(`'a maple` is a shorthand for `'a Maple.t DynaMoW.symb`).

The latter takes a Maple object assumed to be a Maple list and returns an OCaml list of Maple objects. The former performs the opposite operation: it inputs an OCaml list of Maple objects and returns a new Maple object, hiding (and forgetting) that it is a Maple list.

3. Interfacing the Mathematical Web

3.1 Services as Modules

A DynaMoW application is structured around special OCaml modules that we call *services*. Rather than OCaml functions, these are independent components, communicating over the network by serialization and deserialisation. Their general role is the computation of symbolic OCaml values enriched with some presentation format. To this end, a service returns a product of the form `doc * obj`, where type `doc` represents a document that can be serialised and served by a CGI and type `obj` is the type of returned values. Services can call one another, taking decisions based on the symbolic values (`obj` types) so as to recombine the document values (`doc` types).

Services share a lot of (parametrised) logistic code, still, they could not be generated by a mere functor application. Instead, a service is declared through a dedicated keyword `let_service` whose expansion by Camlp4 reveals a large boilerplate. The returned module just exports the input and output types of the service, together with the optional default values of its parameters and functions to call it. As an example, suppose we want a service named `LogInt` to calculate the integral of the nth power of the logarithm function. This can be declared as follows:

```
# let_service LogInt (n : int) : string * maple =
    let res = <:symb< int(log(x)^$(int:n), x) >> in
    (<:string< sprintf("%a", $(res)) >>, res)
```

Here the "document" returned by the service is a `string`, the result of letting Maple pretty print the algebraic quantity by `sprintf`; the next section will explain how more general web documents can be generated.

Often, the pair calculated by a service corresponds to the same symbolic value in two forms: the first intended for presentation to the end user, the second for further processing in OCaml. A more pragmatical view is that `obj` is only that part of the computed symbolic object that is really needed in further calculations. So, for some services and typically those that generate a whole web page rather than a section in a web page, `obj` is just OCaml's `unit`. (Compare files `Definition.ml` and `GeneralFormulaForTaylorBound.ml` in the DDMF source, available from the DDMF web site.)

When calling a service, the caller has access to either output via the functions `obj` and `descr`: Function `obj` immediately computes and returns the value in symbolic form:

```
# LogInt.obj (3, ()) ;;
- : LogInt.obj = << ln(x)^3*x-3*x*ln(x)^2+6*x*ln(x)-6*x >>
```

Function `descr` delays evaluation: it creates a descriptor that can later be used with function `DynaMoW_services.Services.call` to obtain the presentation as a string:

```
# let descr = LogInt.descr 3 None ;;
val descr : LogInt.doc_type DynaMoW_services.service_descr
  = <abstr>
# DynaMoW_services.Services.call descr ;;
- : DynaMoW_services.Services.service_type * string =
  ("Binary", "ln(x)^3*x-3*x*ln(x)^2+6*x*ln(x)-6*x")
```

Most computer-algebra systems leave a lot of side effects in a running session. To achieve some context independence of web services, we chose to make each symbolic session local to one service execution. This immobilises much resource, which made it not desirable to allow for recursive services. (And we have not felt the need for it in practice.) To forbid this, the two functions `obj` and `descr` are not exposed to the service definition.

3.2 Web Documents as Nested Quotations

Web documents are structured, tree-like documents that are usually stored in some XML string representation, often a combination of HTML, MathML, SVG, and similar XML dialects. Each variant language is specific to some kind of data to be displayed—MathML for mathematics, SVG for graphics—, but XML dialects all describe static trees. And all such languages share, by nature, a source of heaviness: the markups used to create a very regularly well-balanced

[12] `http://algo.inria.fr/libraries/`

language. This makes them better suited for automatic document generation than for human generation.

Concerning the generation of dynamic web documents, PHP was our first candidate. But we rejected this choice as it could not accommodate the symbolic code as easily as the document code, and also, we have to admit, for a matter of taste against the programming style. Another option to be considered was JavaScript, which embeds naturally in HTML. JavaScript allows to refer to the document, but in too low a level, and would also not have accommodated the symbolic code nicely.

Just as symbolic programs have been incorporated directly into OCaml code, we wanted to be able to write the pieces of text that make up a mathematical web document as quotations. Additionally, let us remark that some literal mathematics have to be part of the source code, typically to display equations of the form "$f(x) =$ computed value". This could be obtained in DynaMoW by an expression of the form `<:imath<f(x) = $(symb:v)>>`. For the literal part, `f(x) =` in the example, we had to choose a language that a human would easily type, and LaTeX looked like the only reasonable choice: MathML was ruled out by its verbosity. Thus, although we defined few quotation kinds for documents, an added complexity came from more levels of possible nesting, as a consequence of three input kinds that can contribute to texts in documents:

- literal strings, playing a role akin to the "character data" (CDATA) of XML, to write natural-language text;
- literal LaTeX code fragments, which appear as escapes from literal strings to denote a mathematical formula;
- symbolic code fragments, which appear as escapes from LaTeX code to denote a symbolic calculation whose result has to be displayed.

Literal strings with LaTeX escapes are introduced by either of the two quotations kinds `<:par<...>>` and `<:text<...>>`, which correspond respectively: *(i)* to full paragraphs possibly disrupted by displayed mathematical formulas; *(ii)* to linear text used for titles and the like, as well as to portions of sentences that have to receive a special rendering style. Literal LaTeX is introduced by either of the two quotations kinds `<:imath<...>>` and `<:dmath<...>>`, which stand respectively for an inline and for a displayed mathematical formula. Finally, the same notation `<:symb<...>>` as used directly in OCaml allows the inclusion of generated LaTeX, by introducing a symbolic evaluation whose output is converted to LaTeX by the symbolic plugin before presentation. (The plugin will typically use the computer-algebra system for this task if it exports a conversion facility.)

On the implementation side, nesting quotations no longer rely solely on the ability of Camlp4 to detect quoted strings by parsing OCaml. But, fortunately, the analysis of nested quotations does not require more than counting opening and closing parentheses, making the implementation very simple.

In addition to quotations, natural antiquotations allow insertion of simple datatypes in literal text and LaTeX, and content-manipulation functions can next be used to transform or combine the quotation-generated contents. Thus,

```
let l = [2; 3; 5; 7] in let n = List.length l in
List.fold_left
  (fun s a = s @:@ <:par<, $(int:a)>>)
  <:par<There are $(int:n) elements \
    in this list, viz.>> l
```

where `( @:@ )` is a DynaMoW operator to merge the contents of two `<:par<...>>`, results in the display

There are 4 *elements in this list, viz.,* 2, 3, 5, 7

More functions combine paragraphs into lists, sections, etc.

These content quotations in DynaMoW have to be contrasted with the quotations proposed by another OCaml-based tool, OCamlDuce, a general XML transducer and validator: its quotations embed general XML strings, including XML markups (or, provided a DTD is given, this can be a string in some XML dialect). By contrast, the `<:par<...>>` and `<:text<...>>` quotations of DynaMoW allow no HTML markup. Indeed, the choice with DynaMoW is to generate a web document by calling constructors directly on subdocuments, much like using the module `XHTML.M` in Eliom, the OCaml language extension that comes with the web programming framework Ocsigen [2]. A difference is that DynaMoW proposes a restricted set of document constructs, rather than arbitrary HTML markups: the rationale here is to make it easier to achieve the uniformity of presentation over a whole generated web site.

3.3 Mathematical Presentation Online

Beside symbolic computations, *mathematical presentation* is another key component of DynaMoW. At the time of writing, no technical solution is as good as what we would want for our purpose, as the existing tools either provide with no completely nice rendering or are too slow for dynamic use, as they were implemented for static-only presentation.

The oldest method for presenting mathematical formulas is to embed Gif or PNG pictures. Fast conversion from LaTeX is nowadays possible on the server, and we use `latex` and `dvipng` for this task. However, this approach is subject to pixelisation when the user zooms in.

A more modern way is to let the client do the rendering. A nice tool for this is jsMath, implemented in JavaScript. Here, mathematical formulas are embedded directly into the web page, using dedicated markups. On the client, jsMath processes the LaTeX, typesetting it very much like LaTeX would, and using LaTeX mathematical fonts for the display. The rendering is very fine, but JavaScript can lead to unwanted latency and can be slow on some systems[13].

MathML is what is presented as the future, although, at the time of writing, it is reliably supported by Firefox only. The rendering is as fast as non-mathematical text, but it is a bit less nice than a typical LaTeX rendering, owing mostly to too rigid alignments. As LaTeX is the necessary format at an intermediate stage of our translation process, we had to use a converter from LaTeX to MathML. We could find no perfect tool for this:

- Distler's itex2MML (in C) is as fast as we need, but it understands a variant of TeX that we could not produce easily (spaces are needed to denote mathematical products) and cannot easily be extended to understand the $\mathcal{AMS}$-TeX constructs we use;
- Miller's LaTeXML (in Perl) does an excellent job in producing very nice LaTeX, but it is meant to be used offline, and is too slow by a factor 100 to be used for run-time generation of MathML (the slowness is caused by the very general LaTeX parser).

The current version of DynaMoW uses a cocktail of solutions (MathML produced by LaTeXML, jsMath, and PNG produced by `dvipng`) so as to get reasonable results

[13] Internet Explorer, whether 8 or 9, on Windows 7. It is not clear whether this is due to IE itself or to the specific glue of jsMath.

on the main five browser families. At the same time, we continue evaluating MathJax, a new software under very active development, which should supersede our solutions above; MathJax has specific code for Internet Explorer, but still was too slow for our needs at the time of writing.

4. Security and Performance Issues

Security and robustness. Minimal measures prevent simple attacks on the server: parameter passing between services is secured by authentication to prevent injection of symbolic code; an instance of a computer-algebra system will be given limited time and memory resources to avoid denial of service.

Scalability and efficiency. A large effort, both in the design of DynaMoW and in the implementation of our applications, has been put to realise the *statelessness* of each service. This allows the replication of servers, and, as a bonus, independent page fragments are loaded asynchronously and can be calculated in parallel, with no need for any explicit syntax when programming: the right server configuration automates parallelism provided the programmer implements in a sufficiently modular way. This is well exemplified by the file `SpecialFunction.ml` in the DDMF source, which encodes the skeleton of each article in DDMF: each of the eight calls to `DC.inline_service` results in a different section in the web page, which is computed asynchronously.

With respect to our server architecture, DDMF and ECS use for now a simple model with one instance of a computer-algebra system per DynaMoW instance, setting up several DynaMoW instances behind a web server. In the future, we plan to have a single DynaMoW instance with potentially many instances of the same computer-algebra system and a cache for both documents and symbolic values (caching outputs from each service, keyed by the inputs). This will counterbalance the need for repeating some calculations in order to maximise independence of page fragments.

Still, computing a full fresh web page can require a dozen seconds of sequential calculations, primarily because of complex symbolic calculations. To mitigate this, we rely heavily on the web-server cache for better performance, which proves beneficial especially when the site is visited by web crawlers.

Text-based communication with the computer-algebra system is the main bottleneck. Using an API like OpenMaple will give better performance but is more system specific.

Client-side calculations One could think to take advantage of client-side resources for both scalability and efficiency. A typical general-purpose computer-algebra system is too large an application to be transmitted from server to client (not to mention license issues). A reimplementation in, say JavaScript, seems unfeasible, even for a limited set of functionality, and could prove to be too slow. Additionally, it would hardly be portable, as linking to C libraries would be necessary (at least for arbitrary-precision integers). A more appealing approach would be to run another instance of a DynaMoW plugin on the client side, with the computer-algebra system installed on the client.

5. Conclusions

Development and debugging process. The source of DDMF shows two extremes of coding styles: one with long symbolic quotations and no call to any external symbolic library, another with short quotations to a large external library. (Compare files `DiffeqToRecurrence.ml` and `TaylorExpansion.ml` in the DDMF source.) The first case is needed for an intrusive instrumentation of preexisting Maple libraries. With time, we want to show more certificates of the computed data on the web site, which will lead to more instrumented Maple code and increase the proportion of quotations. To modify code, we either modify the external Maple library or the OCaml code, or both. Introducing an adequate ocamltop that is able to call Maple and display (external) symbolic values made the debugging much easier.

Advantages of functional style. Using a language that favours functional style had two main advantages:

First, it made it very easy to deal with the content trees representing web documents. By making us have a recursive representation of the final document, we could now easily implement a translator to a different document format (say, PDF). By contrast, in an earlier implementation in imperative style, we just printed HTML along the execution of our symbolic calculations.

Second, and maybe more importantly, it incited and helped us to refactor the symbolic code in more independent functions. This made the interactivity of the site and the reactivity of the server easier to achieve, by enabling incremental and/or distributed calculations of independent parts.

Types in antiquotations. The main torment in the prototyping phase concentrated around the adequacy of Camlp4 to our project, to the point that it has not been clear immediately that DynaMoW should be an OCaml library. Indeed, for our first prototype (used before DDMF v1.5), we wrote an (OCaml) interpreter for a "BASIC with quotations and antiquotations". In the end, the main drawback with the final choice is that Camlp4 does not offer type analysis at preprocessing-time. This leads to too much type annotation. The following example makes this clear:

```
let a = 3 in <:symb< f($(int:a)) >>
```

If the preprocessing stage done by Camlp4 was aware of types, we could simplify `$(int:a)` to `$(a)`. Our source of DDMF proves that a typical application will have many such annotations. We could partially alleviate this by choosing default antiquotation types, but ironically enough, such annotations were not needed in our first "BASIC" prototype. It turns out that we chose Camlp4 as it provides syntax extensions and quotations in the same tool, but there clearly is room for a quotation/antiquotation extension tool that could infer type information and perform different calculations based on it.

Declarations of services: Functor application is not enough. The main reason why services require a language extension and cannot be obtained just by a functor application is the need for serialisation and deserialisation of the parameters exchanged between services via query strings in URLs. As the number and types of parameters vary, we could generate serialisation and deserialisation functions only on the meta-level, using Camlp4. For comparison sake, the Eliom module `Eliom_parameters` also provides a representation for the GET parameters of a service, and also requires a language extension. A difference is that services are not intended to be called from the OCaml code (with the exception, to some extent, of the use of the function `Eliom_services.preapply`). Service declarations and calls in DynaMoW resemble function calls, whereas declarations of service parameters in Eliom are done by providing the parameter names as strings together with a rep-

resentation of their types to a service registering function (`Eliom_services.new_service`).

A nice side effect of the language extension for service declaration in DynaMoW is also to avoid repetitions in boilerplate (argument names, etc.).

Mathematical rendering. It seems our applications naturally generate web pages with larger formulas than what the online presentation tools are targeting. This made it difficult for us to find the right trade-off between a nice and a fast mathematical rendering (see section 3.3).

In particular, even jsMath and its successor MathJax provide too slow rendering on certain systems. But we had no time to test the version 1.1 of MathJax, which has been released very little time before the present submission, and promises "a number of performance improvements".

Interfacing with Ocsigen. We have not used Ocsigen in DynaMoW as it provides with more general XML than what we wanted to expose to our programmers (see end of §3.2)[14]. Thus, we still rely on an external general-purpose web server (Apache or lighttpd), with DynaMoW a FastCGI[15]. We want to explore the possibility of making DynaMoW a module for Ocsigen and expect added efficiency by doing so.

Expressivity of interactivity. Although the ability of Sage with respect to web display is merely to display classical symbolic sheets using HTML and JavaScript, in a way that is not intended for an all-user web site, Sage contains a few nice interaction functionalities. Particularly so is the `@interact` function, which from the user's point of view presents the input of a parameter in a range as a draggable slider. Besides being nice, this solves the question of soundness checking of the input data. In the same vein, the interaction with parameters limited to a (closed) enumeration type could be rendered better than by using textual fields. We plan to develop such aspects in the future.

(Symbolic) Code extraction. Early in the development of the project, we realised that instrumenting a symbolic library under continued development would lead to the need of double maintenance. Indeed, we need to keep a version of the library for computations off the web. We have prototyped the extraction of pure Maple libraries from our OCaml-with-Maple application just enough to realise that a complete code extraction would imply automatically undoing some of the paradigm-changing port described in section 2.4. This automation is not mature yet and will be the subject of future work.

Acknowledgments

The conception of DynaMoW has benefited greatly from discussions with James Leifer, Marc Mezzarobba, Nicolas Pouillard, and Didier Rémy. As DynaMoW was initially developed as the underlying engine of DDMF, it has also been influenced a lot by discussions with DDMF developers.

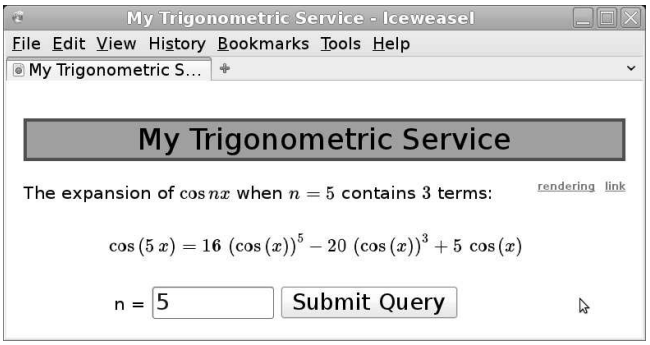
References

[1] M. Abramowitz and I. A. Stegun, editors. *Handbook of mathematical functions*. Dover, New York, 1992.

[2] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a web programming framework. In A. Tolmach, editor, *ICFP'09*, pages 311–315, 2009.

[14] and for unclear license issues

[15] `http://www.fastcgi.com/`



File `Trigo.ml`:

```
use_cas Maple
type_symb maple = unit Maple.t DynaMoW.symb

let_service Trigo (n : int = 2) :
  DynaMoW_services.Content.sec_entities * unit =
  let m_expr = << cos($(int:n) * x) >> in
  let s = Expand.obj (m_expr, ()) in
  let par_intro =
    <:par<The expansion of <:imath<\cos nx>>
      when <:imath<n = $(int:n)>>
      contains <:imath<$(int:s)>> terms:>>
  and par_math =
    DynaMoW_services.Content.inline_service
      (Expand.descr m_expr None) in
  let par =
    DynaMoW_services.Content.(@@@) par_intro par_math in
  (DynaMoW_services.Content.section
     <:text<My Trigonometric Service>> par,
   ())
```

File `Expand.ml`:

```
use_cas Maple
type_symb maple = unit Maple.t DynaMoW.symb

let_service Expand (expr : maple) :
  DynaMoW_services.Content.sec_entities * int =
  let s = << expand($(expr)) >> in
  (<:par<<:dmath<$(symb:expr) = $(symb:s)>>>>,
   <:int< nops($(s)) >>)
```

Table 1. Code for a simple service `Trigo` that calls a simple service `Expand` and, at the top, the resulting display. Proper configuration makes it available from the URL `http://.../example?service=Trigo&rendering=jsMath&n=5`. After entering the default web page (without final `&n=5`), which rendered the case $n = 2$, we changed the value to 5 in the form so as to get the display above.

[3] A. Benoit, F. Chyzak, A. Darrasse, S. Gerhold, M. Mezzarobba, and B. Salvy. The Dynamic Dictionary of Mathematical Functions (DDMF). In *The Third International Congress on Mathematical Software (ICMS 2010)*, volume 6327 of *Lecture Notes in Computer Science*, pages 35–41, 2010.

[4] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ML. In *Proc. of the Workshop on ML and its applications*, 1994.

[5] N. J. A. Sloane and S. Plouffe. *The Encyclopedia of Integer Sequences*. Academic Press, 1995.