DynaMoW User's Manual

(for DynaMoW release 1.0.0)

Frédéric Chyzak and Alexis Darrasse

September 9, 2011

Contents

Status of this document. This user's manual is intended for DynaMoW 1.0.0. It is partly unfinished, but we thought it is more useful to have some incomplete documentation now than wait for who knows how long. Intended but not written section will be marked as "To be written."

1. About DynaMoW

1.1. Why Dynamic Mathematics on the Web? The presentation of mathematical contents on the web has become nicer and nicer over the last years, with tools like jsMath, which relies on the widespread deployed JavaScript, or MathML, which should soon be well interpreted by most of the web browsers. By introducing scalable fonts, both approaches result in a much better rendering than the older method of displaying mathematical formulas as images.

But the presentation of mathematics on the web is essentially always static: a mathematical web site is often not much more than a book typeset for the web. As an exception, animated jpeg go a bit beyond that, for example for animated plots, but they present a statically defined succession of mathematical objects.

This static nature of common presentation is a real limitation, as a lot of the presented mathematical objects are infinite in nature: As only a finite truncation of them can be displayed at a time, in a static model, only a statically chosen limited amount of information can be displayed. Typical examples are ranges of plots, orders in series expansions, precisions in numerical calculations, and so on.

This situation is ignoring that the web also allows for interactions with documents: readers should naturally be able to ask for interactive modifications of the parameters used in the generation of formulas, and for the dynamic recalculations and redisplay of the mathematical objects. This, of course, relies on the ability to perform run-time mathematical calculations, and computer-algebra systems (CAS) are a suitable tool for this.

DynaMoW fulfills these needs: it is a tool to simultaneously interact with mathematical readers, monitor calculations in computer-algebra systems, and generate or regenerate mathematical web pages to be displayed.

Additionally, from the point of view of the designer of a web site with dynamic mathematical content, several layers of programming languages need to coexist in the application: the monitoring/controlling language, one or several computer-algebra languages, a language to express web documents. It is natural to require that those layers appear as naturally as possible in the same source files, and DynaMoW takes care of this as well.

1.2. Overview of DynaMoW. DynaMoW is implemented as a language extension of OCaml[1], together with a relevant system. The most visible feature in this extension is the use of quotations, that is, the use of lexical markups to embark fragments of other languages as is. This is used to embark fragments of computer-algebra sources or fragments of web documents directly into the OCaml source. Additional, antiquotations permit variables to migrate from one layer to the other:

[1] `http://caml.inria.fr/ocaml/`

antiquoted OCaml variables can be used in quotations, either for symbolic[2] values to take part in computer-algebra calculations, or for symbolic values to appear in a mathematical display.

DynaMoW is intended to be independent of the chosen CAS. Here, a CAS is meant as what could be called a "symbolic shell": CAS usually are interpreted systems, prompting users for mathematical commands in imperative style. For DynaMoW to interact with such shells, each symbolic calculator is introduced by a well specified interface, taking the form of a plugin. As of the time of writing, only a plugin for Maple has been tested extensively,[3] but more are available on an experimental level, and more could be developed by users.

Thus, a DynaMoW-based application appears as a sort of compiled OCaml application. Quotations and antiquotations are resolved at a preprocessing stage of the compilation: a Camlp4[4] filter converts the DynaMoW source to pure OCaml source, in which quoted fragments now appear as strings, and antiquotations take the form of suitable strings concatenations and function calls. The obtained OCaml source is compiled next, yielding a usual OCaml stand-alone application. Finally, this application is typically made available online as a CGI (or, rather, a Fast CGI[5]), which monitors run-time interpreted symbolic calculations in a CAS.

1.3. 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 to serve as the underlying engine of the Dynamic Dictionary of Mathematical Functions[6] (DDMF), it has also been influenced by discussions with DDMF developers.

The authors of DynaMoW are Frédéric Chyzak[8] (`frederic.chyzak@inria.fr`) and Alexis Darrasse[9] (`alexis.darrasse@lip6.fr`).

Running DynaMoW requires:

- some version of John Resig's jQuery, which is not provided with the distribution of DynaMoW but is distributed from `http://jquery.org/`[10] under the dual license MIT or GPL Version 2;
- some version of Davide P. Cervone's jsMath, which is provided beside the distribution of DynaMoW and is as well distributed from `http://www.math.union.edu/~dpvc/jsMath/`[11] under the Apache License 2.0.

1.5. Status of the API. Concerning the API, we intend to follow semantic-versioning recommendations. As of 1.0.0, the stable API consists of:

- the keywords `use_cas`, `let_service`, `type_symb`;

[2]Throughout this document, "symbolic" means "related to an external computer-algebra system."

[3]As a consequence, all examples calling a computer-algebra system in this document use Maple.

[4]`http://brion.inria.fr/gallium/index.php/Camlp4`

[5]`http://www.fastcgi.com/drupal/`

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

[7]`http://www.cecill.info/`

[8]`http://algo.inria.fr/chyzak/`

[9]`http://www.ortsa.com/darrasse/`

[10]`http://jquery.org/`

[11]`http://www.math.union.edu/~dpvc/jsMath/`

- the syntax and semantics of quotations and of corresponding antiquotations, as described in §2.2 and §3.2.2, but not their implementation;
- the module `DynaMoW` and its submodules `DynaMoW.Services` and `DynaMoW.FCGI.mli`, as defined in `DynaMoW.mli`.

Are still considered unstable or meant to disappear:

- the modules `DynaMoW.CAS` and `DynaMoW.Internal`.

Out of the existing computer-algebra plugins, `Maple` (as used by `use_cas Maple`) may be considered stable (but not its mplementation); all other plugins are still experimental.

The list of DynaMoW-specific exceptions and their meaning are not fully stable, but the spirit of the implementation and the current status is described in §5.

1.6. Installation and compilation. The latest release of DynaMoW can be downloaded from `http://ddmf.msr-inria.inria.fr/DynaMoW/`[12], where the reader will also find help about installing dependencies. After uncompressing the archive, the script `install-dependencies.sh` will install the author side of jsMath as well as jQuery in the DynaMoW directory.

More dependencies like OCaml, Camlp4, Fast CGI, etc., should be well packaged on most systems. For Debian (and Ubuntu), the script `config_helper/debian` will install all of them; for a recent OS X on which MacPorts has been set up, the script `config_helper/osx` will do.

Then, `make` should just compile DynaMoW to the end, provided you have the needed computer-algebra systems in your path. The file `config.mk.doc` details further compilation options, used to write an optional `config.mk` file.

As interaction with a web server is more delicate and only makes sense once some DynaMoW-based application has been implemented, we postpone final set up to §4.1.

2. Computer-algebra system calls

2.1. Configuration. After configuration, the DynaMoW layer in OCaml has to be initialised by designating the computer-algebra system to be used and loading the run-time configuration file (the contents of this file will be described in §4.7.2).

```
# DynaMoW.init () ;;
- : unit = ()
# use_cas Maple ;;
module DynaMoW_CAS__ :
  sig
    type 'a t = 'a Maple.t
    type cas_code = string
    val evaluator_symbolic : cas_code -> 'a t DynaMoW.symb
    val evaluator_latex : cas_code -> DynaMoW.CAS.latex
    val evaluator_unit : cas_code -> unit
    val evaluator_int : cas_code -> int
    val evaluator_bool : cas_code -> bool
    val evaluator_string : cas_code -> string
    val serialization_of_t : 'a t DynaMoW.symb -> string
    val cas_ref_of_t : 'a t DynaMoW.symb -> string
    val pretty_code_of_t : 'a t DynaMoW.symb -> DynaMoW.CAS.pretty_code
    val latex_of_t : 'a t DynaMoW.symb -> DynaMoW.CAS.latex
```

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

```
    val cas_code_of_bool : bool -> cas_code
    val reset : unit -> unit
  end
# << 2*x+1 >> ;;
- : '_a DynaMoW_CAS__.t DynaMoW.symb = <abstr>
```

The order of these two operations is irrelevant. By calling again `use_cas` it is possible to change the computer-algebra system used. Warning: the type system prevents using variables of the old computer-algebra system until we switch back to it with a new call to `use_cas`.

When working on the toplevel, one can install a pretty-printer of symbolic values:

```
# #install_printer Maple.pp ;;
# << expand((x+1)^2 - x^2) >> ;;
- : '_a DynaMoW_CAS__.t DynaMoW.symb = << 2*x+1 >>
```

2.2. Symbolic quotations and their antiquotations. The following quotations and corresponding antiquotations allow for the simplification of many common patterns in code using the DynaMoW library to call a computer-algebra system. All quotations below take an expression in the internal language of the current CAS. What differs is their output format: whether an OCaml handle to an externally-living symbolic value, or, after some kind of conversion, a value of some native OCaml type.

`<:symb<...>>`:
: Create a reference to a value living in the CAS (the type of the result is `t DynaMoW.symb`). This is the default quotation in DynaMoW-based OCaml code, and it can be abbreviated `<<...>>`.

`<:unit<...>>`:
: Evaluate the code without considering its result. This is mainly useful for declarations that do some side effect in the CAS.

`<:latex<...>>`:
: Return a string containing the LaTeX representation of the expression it was given.

`<:int<...>>`:
: If the expression evaluates to an integer, this quotation returns the corresponding OCaml `int`, otherwise a `DynaMoW.Invalid_type` exception is raised.

`<:bool<...>>`:
: If the expression evaluates to a boolean, this quotation returns the corresponding OCaml `bool`, otherwise a `DynaMoW.Invalid_type` exception is raised.

`<:string<...>>`:
: If the expression evaluates to a string, this quotation returns the corresponding OCaml `string`, otherwise a `DynaMoW.Invalid_type` exception is raised.

The design of the <:int<...>>, <:bool<...>>, and <:string<...>> quotations is based on the informal assumption that the computer-algebra system has types that correspond to int, bool, and string. However, there are cases where the OCaml type is only an approximation of the external type. This is typically so, for example, when a computer-algebra system has integer values that cause an overflow as an OCaml integer. In such cases, the quotation will raise an DynaMoW.Out_of_range exception.

One can inject OCaml values into the CAS expression inside a quotation by using one of the following antiquotations: $(symb:...), $(int:...), $(bool:...), and $(str:...). Each of these takes an OCaml expression that must be of the type designated by the antiquotation name. In all quotations above, the $(symb:...) antiquotation can be abbreviated $(...).

```
# let s = <:int< 1 + 2 + 3 + 4 + 5 >> ;;

val s : int = 15

# s/2 ;;

- : int = 7
```

For another example on how to use these quotations, see the file doc/webless/webless.ml.

The notation just described is defined in the DynaMoW_filter Camlp4 module. It is translated at preprocessing time into code that depends on DynaMoW.CAS and DynaMoW.Internal. A manual, more advanced range range of possible interactions with the CAS is described by the DynaMoW.CAS.CAS_TYPE module type (but it is still unstable).

2.2.1. *A warning about syntax.* Most of the examples of this document use a lot of spaces in OCaml code. It is not our intent here to say what is proper style or not, but a potentially bad interaction of DynaMoW quotations and antiquotations with the OCaml lexer has to be commented on.

The following example shows the use of a default quotation and a default antiquotation, both defaults being symbolic:

```
# let a = << (1+x)^5 >> ;;

val a : '_a DynaMoW_CAS__.t DynaMoW.symb = << (1+x)^5 >>

# << $(a)^2 >> ;;

- : '_a DynaMoW_CAS__.t DynaMoW.symb = << (1+x)^10 >>
```

The trouble comes when one rejects spaces and writes:

```
# <<$(a)^2>>;;

Error: Parse error: illegal begin of top_phrase
```

The explanation of this error is that the OCaml lexer looks for the longest possible lexical entities, so that <<$(is lexed as <<$ followed by (, rather than the wanted << followed by $(. If the wanted style is minimalistic with respect to spaces, the programmer has to write:

```
# <:symb<$(a)^2>>;;

- : '_a DynaMoW_CAS__.t DynaMoW.symb = << (1+x)^10 >>
```

2.3. Interface DynaMoW.mli.

`val init : ?config_file:string unit -> unit`:
The `DynaMoW.init` function initializes DynaMoW by reading the configuration file. By default, configuration is read from file `dynamow.conf` in the current working directory.

`val register_configuration_section : string -> string list -> ((string * string) list -> unit) -> unit`:
An application that need some configuration will have a relevant section in the configuration file, and will register the corresponding variables through `register_configuration_section sec options callback`. Here, `sec` is the name of the section, `options` is the list of recognized variable names for the section, and `callback` is a function that takes an association list of (recognized) names to values and deal with to perform the relevant initializations.

`type 'a symb`:
Symbolic values have type `DynaMoW.symb`. The type parameter encodes the corresponding computer-algebra system. For example, Maple symbolic values have type `'a Maple.t DynaMoW.symb`. This second level of parametrisation can be used to reflect in a finer way in OCaml different datatypes returned by the same CAS.

`exception Invalid_type of string * string`:
When the evaluation of some symbolic code by an `evaluator_...` of a plugin does not result in a value of the expected type, an exception `DynaMoW.Invalid_type (w, c)` is raised, where `w` encodes the wanted type and `c` is the computed symbolic value.

`exception Out_of_range of string * string`:
When the evaluated symbolic code results in a value of the expected type but out of the range of values of the corresponding OCaml type, an exception `DynaMoW.Out_of_range (w, c)` is raised, where `w` encodes the wanted type and `c` is the computed symbolic value.

2.4. CAS Plugins. Computer-algebra systems can be used with DynaMoW through dedicated plugins. Currently predefined plugins are a well tested Maple plugin, a simple OCaml plugin that serves as an example, as well as experimental Mathematica and Sage plugins. These plugins (found in the directory `src/cas/`) are compiled at the time DynaMoW is compiled; user-implemented plugins can also be used, provided they abide by the relevant interface.

Concerning the signature of plugins, the interface `DynaMoW.CAS.mli` is still considered unstable, except for the module type `DynaMoW.CAS.CAS_TYPE` describing computer-algebra plugins, which is meant to be almost stable. A module is of type `DynaMoW.CAS.CAS_TYPE` if it contains the following elements:

`type 'a t`:
A phantom type that is used as a parameter to the `DynaMoW.symb` type for every value produced by the plugin. This way, the typer ensures that values coming from one plugin cannot end up in another one. One can use the parameter of type `t` to give hints to the OCaml typer about the types of values in the CAS (as illustrated in **??**).

`type cas_code = string`:
Commands that are to be interpreted by the CAS are for now stored in OCaml strings, but this type will become opaque in the near future.

`val evaluator_symbolic: cas_code -> 'a t DynaMoW.symb`:
Evaluate the CAS code and return a reference to the result. Used typically when evaluating quotations `<:symb<...>>`.

`val evaluator_latex: cas_code -> latex`:
Evaluate the CAS code and return the LaTeX representation of the result. Used typically when evaluating quotations `<:latex<...>>`.

`val evaluator_unit: cas_code -> unit`:
Evaluate the CAS code. Used typically when evaluating quotations `<:unit<...>>`.

`val evaluator_int: cas_code -> int`:
Evaluate the CAS code and return the OCaml `int` corresponding to the result. Used typically when evaluating quotations `<:int<...>>`. If the result is not an integer, a `DynaMoW.Invalid_type` exception is raised; if the result does not fit in an OCaml `int`, a `DynaMoW.Out_of_range` exception is raised.

`val evaluator_bool: cas_code -> bool`:
Evaluate the CAS code and return the OCaml `bool` corresponding to the result. Used typically when evaluating quotations `<:bool<...>>`. If the result is not a boolean, a `DynaMoW.Invalid_type` exception is raised; if the result does not fit in an OCaml `bool`, a `DynaMoW.Out_of_range` exception is raised.

`val evaluator_string: cas_code -> string`:
Evaluate the CAS code and return the OCaml `string` corresponding to the result. Used typically when evaluating quotations `<:string<...>>`. If the result is not a string, a `DynaMoW.Invalid_type` exception is raised; if the result does not fit in an OCaml `string`, a `DynaMoW.Out_of_range` exception is raised.

`val serialization_of_t : 'a t DynaMoW.symb -> string`:
Return a compact string representation of a symbolic value living in the CAS. The result might not be human-readable, but it can later be evaluated in a fresh instance of the CAS to recreate the original value. This function is used internally for argument passing between services.

`val cas_ref_of_t : 'a t DynaMoW.symb -> string`:
Return a string containing a reference to a symbolic value living in the CAS. The result can later be evaluated in the same instance of the CAS to get a new reference to the original value. This function is used internally to implement the `$(symb:...)` antiquotation.

`val pp : Format.formatter -> 'a t DynaMoW.symb -> unit`:
This pretty-printer can be used as an argument to the `#install_printer` toplevel directive to make symbolic values more transparent.

`val pretty_code_of_t : 'a t DynaMoW.symb -> pretty_code`:
Return a human-readable representation of a symbolic value living in the CAS. The end user should be able to use it to recreate the original value.

`val latex_of_t : 'a t DynaMoW.symb -> latex`:
Return a LaTeX representation of a symbolic value living in the CAS.

`val cas_code_of_bool : bool -> cas_code`:
Transform an OCaml boolean to CAS code that evaluates to the corresponding boolean value in the CAS.

`val reset: unit -> unit`:
Reset the CAS to a fresh state.

To simplify creation of new plugins, we provide the `DynaMoW.CAS.Shell.Generic` functor that should be well adapted to most CAS with a command-line interface providing a read-eval-print loop. This functor takes as input a module that contains a string for each CAS command needed by DynaMoW. This interface is not yet described in detail as it is unstable.

3. Web services and web document content

A DynaMoW application is structured around services, which are special OCaml modules to be described in this section. The typical role of a service is the computation of some (unrestricted) OCaml value, more often that not involving symbolic types, 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).

DynaMoW applications base on a small number of predefined document kinds to represent text files, binary blobs, SVG images, and web document trees. Such document contents are also described in this section, together with operations on them.

3.1. Web services.

3.1.1. *Predefined service types.* To be written: These are `content_service`, `svg_service`, and `[< service_type ] service_descr`, etc.

3.1.2. *Service declaration.* A DynaMoW service is declared through a dedicated keyword `let_service`. This returns a special OCaml module of the following generated type:

```
module type SERVICE_TYPE = sig
  type required
  type optional
  type doc_type
  type obj
  val defaults : optional
  val obj : required * optional -> obj
  val descr :
    required -> optional option ->
    doc_type DynaMoW.Services.service_descr
end
```

Here all types are built from the service-declaration syntax to be described below.

More specifically, types `required` and `optional` are tuples describing the required and optional parameters of the service. Required parameters can be of any type, while optional parameters may only be OCaml integers, booleans, or strings. Support for more types, especially enumerations, will be added in future versions.

Let us describe the `let_service` syntax on an example. Suppose we want a service named `LogIntegral` to calculate the integral of the nth power of the logarithm function. This can be declared as follows:

```
type maple = unit Maple.t DynaMoW.symb
let_service LogIntegral (n : int) : string * maple =
  let res = << int(log(x)^$(int:n), x) >> in
  (<:string< sprintf("%a", $(res)) >>, res)
```

(The resulting output is provided in the appendix. For a detailed understanding of it, not to be aimed at at a first read, see §3.4.) The service name is followed by the list of parameters, here a single required parameter, the integer `n`. Optional parameters are signaled by specifying a default value. For example,

```
let_service LogIntegral (n : int = 1) : string * maple =
  ...
```

would allow calling the service with no explicit value for n.

The list of parameters is ended by a colon followed by a product `doc * obj` of types describing the service output and, after an equal sign, the service body that has to be an OCaml expression of this type.

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 to further processing in OCaml. 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:

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

On the other hand, 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 = LogIntegral.descr 3 None ;;
val descr : LogIntegral.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")
```

Here, the actual presentation form is the second element of the pair, the first one being an encoding of the kind of service. For the type system, this kind is represented through the generated type `doc_type`, and the parametrisation of the return type of `descr` ensures no kind clash when services call one another.

In more complex applications, the presentation form is not necessarily a string, but some built-in serialisable structure, as we will see in §3.2.2.

Comments on low-level structures introduced by a service declaration are given in §3.4.

3.2. Construction of content.

3.2.1. *Predefined content types.* DynaMoW predefined types for document trees are `sec_entities`, `par_entities`, and `text_entities`. Objects of the former type are constructed via specific functions calls; objects of the two latter types are usually constructed via quotations, then transformed or combined by functions. The relevant functions and quotations are described below.

The type `sec_entities` is the highest level of nesting in a page; it is used to encapsulate a collection of entities to be displayed in vertical succession, like a sequence of sections or a sequence of paragraphs; such vertical successions approximately correspond to sequences of `<div>`s in HTML documents.

The type `par_entities` encapsulates a collection of lexical elements that constitute a paragraph, that is a collection of entities to be displayed in horizontal

succession; such horizontal successions approximately correspond to sequences of `<span>`s in HTML documents. However, the treatment of displayed formulas is a notable exception: it disrupts the horizontal flow of text, but displayed formulas appear in a `par_entities` as they are part of the logical flow of sentences.

The type `text_entities` encapsulates a collection of lexical elements that constitute a sentence; its constituents are textual fragments, possibly decorated with some style, and inlined mathematical fragments.

The entity datatypes above are nested in various ways, subject to the following constraints:

- values of type `sec_entities` are sequences of sections, lists, and paragraphs;
- the root of a content document, the topmost `sec_entities`, should contain a single section.

Some of the constructs described below can be applied to the three predefined datatypes above. To make this possible, these three types are implemented as synonymous for specialisations of a parametrised type `'a entities`.

3.2.2. *Content quotations and their antiquotations.* The following quotations and corresponding antiquotations are the primary means to denote values of the "horizontal" types, `par_entities` and `text_entities`, allowing for the simplification of many common patterns in code using the DynaMoW library.

`<:par<...>>`:
Create a paragraph from some text. Caution: this text is restricted to UTF-8 strings; it is not allowed to involve any of the `&...;` entities that appear in parsed character data (PCDATA) of XML. Antiquotations and nested quotations are possible in order to include more complicated content:

- One can use the `<:imath<...>>` quotation to include an inlined mathematical expression, in the same way one would use `$...$` in a LaTeX document. Inside such a nested quotation, the antiquotations `$(symb:...)`, `$(int:...)`, `$(bool:...)`, and `$(str:...)` are available to include values of OCaml variables. The `<:symb<...>>` or `<<...>>` quotation can be used inside `<:imath< ... >>` to evaluate a CAS expression.
- The `<:dmath<...>>` quotation does the same thing as `<:imath<...>>`, but is rendered as displayed maths.
- The `<:isymb<...>>` and `<:dsymb<...>>` quotations provide shortcuts for `<:imath<<:symb<...>>>>` and `<:dmath<<:symb<...>>>>`, respectively.
- The antiquotation `$(string:...)` allows for the result of an arbitrary OCaml string expression to be included in the paragraph. The antiquotations `$(int:...)` and `$(bool:...)` do the same, the resulting value being passed through functions `string_of_int` and `string_of_bool` respectively.
- The antiquotation `$(p_ent:...)` (abbreviated `$(...)` allows for calculated `par_entities` to be included in a paragraph. Similarly, `text_entities` can be included through the `$(t_ent:...)` antiquotation.

`<:text<...>>`:
Create a `text_entities` value. Except for its output type, is similar to `<:par<...>>`. This quotation is most useful for creating titles

and the like. Only the relevant subset of quotations and antiquotations available in `<:par<...>>` can be used inside a `<:text<...>>`, namely: `<:imath<...>>`, `<:isymb<...>>`, `$(symb:...)`, `$(int:...)`, `$(bool:...)`, `$(str:...)`, and `$(t_ent:...)`. The `$(t_ent:...)` antiquotation can be abbreviated `$(...)`.

`<:imath<...>>`:
Create a `par_entities` containing inlined maths than can later be injected into a `<:par<...>>` by using the `$(p_ent:...)` antiquotation.

`<:dmath<...>>`:
Same as `<:imath<...>>`, but for displayed maths.

The notation just described is defined in the `DynaMoW_filter` Camlp4 module. It is translated at preprocessing time into code that depends on `DynaMoW.CAS`, `DynaMoW.Services`, and `DynaMoW.Internal`.

3.2.3. *Content-manipulating functions.* The functions described in this section allow to create and combine entities to create structured documents.

The following functions and constant are used to concatenate and merge entities of the same type:

`val ( @@@ ) : 'a entities -> 'a entities -> 'a entities`:
Concatenate two entities of the same kind. For example, if `s1` is a `sec_entities` containing two paragraphs and `s2` is a `sec_entities` containing three paragraphs, `s1 @@@ s2` is a `sec_entities` with five paragraphs.

`val ent_null : 'a entities`:
Neutral element for the `@@@` operation.

`val ( @:@ ) : sec_entities -> sec_entities -> sec_entities`:
Merge two paragraphs. Both arguments should contain a single paragraph (i.e., be the result of either the `<:par<...>>` quotation or of the function `( @:@ )`) and the result will also contain a single paragraph.

Sections and list are created through the following functions:

`val section : text_entities -> sec_entities -> sec_entities`:
A call `section t b` builds a section with title `t` and body `b`. The body has to be either a sequence of sections or a sequence of paragraphs and lists.

`val ordered_list : sec_entities list -> sec_entities`:
A call `ordered_list items` builds an enumeration where items are the elements of `items`. The list `items` may not contain sections.

`val unordered_list : sec_entities list -> sec_entities`:
A call `ordered_list items` builds an bulleted list where items are the elements of `items`. The list `items` may not contain sections.

One can include or link to the result of other services, or an external resource through the following functions:

`val include_service : content_service -> sec_entities`:
Include the content of another service. The content is not loaded until the user asks for it; only the title is displayed.

`val inline_service : content_service -> sec_entities`:
Include the content of another service. The content is automatically loaded and displayed.

`val plot : svg_service -> par_entities`:
Include an SVG image produced by some service. The image is inlined.

`val link : string -> text_entities -> par_entities`:
A call to `link url text` creates a link to the external url `url`. The text `text` materializes the link.

`val link_service : [< service_type ] service_descr -> text_entities -> par_entities`:
A call to `link_service descr text` creates a link to the service with descriptor `descr`. The text `text` materializes the link.

The DynaMoW library also provides functions for applying some decoration on a text fragment or paragraph.

`val code : string -> text_entities`:
A call to `code s` will create a text fragment containing `s` typeset as code. Generally, this corresponds to a fixed width font.

`val definition : string -> text_entities -> text_entities`:
A call to `definition "term" def` will include the text fragment "term" in the running text, associated with a tooltip containing `def`.

`val css_of_string : string -> css`:
Create a value of type `css` from a string. The resulting value can be used to apply an arbitrary style to text fragments or paragraphs. The argument must contain a well-formed fragment in standard CSS syntax.

`val dstyle : css -> string -> text_entities`:
Create text with an arbitrary style.

`val note : sec_entities -> sec_entities`:
Transform a paragraph to a note. This is generally styled in smaller font.

`val warning : sec_entities -> sec_entities`:
Transform a paragraph to a warning. This is generally styled in orange color.

`val pstyle : css -> sec_entities -> sec_entities`:
Apply an arbitrary style to a paragraph.

3.2.4. *Calling and linking to a service.* One uses the descriptors obtained through the `descr` function of a service to make the result of this service accessible in a document. The DynaMoW library provides with different presentation options: one can either include a link or inline the result, and the inlining operation can be performed either immediately or asynchronously. The options available depend on what kind of document the service creates:

- for an arbitrary service, one can only create a simple link through the `link_service` function;
- an SVG-producing service can either be linked or inlined through the `plot` function;
- the `inline_service` and `include_service` functions are reserved for services that produce content documents.

3.3. Extra service parameters. To be written: The construct `with` introduce extra service parameters, of less frequent use, like specific web-page titles or CSS styles.

3.4. Description of generated modules and types around a service. This section deals with the advanced topic of the structures generated by a service declaration. These structures are not meant to be used directly by the programmer, but understanding them is needed occasionally, especially to understand the output printed by a `let_service` on the toplevel, and weird-looking error messages that originate from type mismatches.

For a complete understanding, we proceed to describe the introduction of a service named `Bar` by `let_service Bar ...` in a file `Foo.ml`. The reader will find it useful to read Appendix A in parallel, thinking that the service/module name `LogIntegral` plays the role of `Bar`.

The main effect of declaring a service `Bar` is the creation of a module `Foo.Bar` that can be manipulated like in §3.1.2, that is, essentially, through the two functions `Foo.Bar.descr` and `Foo.Bar.obj`. The module `SERVICE_Bar__` and signature `Bar_TYPE__` are intermediate constructs, needed internally for the creation of module `Foo.Bar`. Users should ignore them, except on an informative level for the types declared in the signature of `SERVICE_Bar__`:

- type `required` is the Cartesian product of the types of mandatory formal arguments of the service;
- type `optional` is the Cartesian product of the types of optional formal arguments of the service;
- type `doc_type` encodes the concrete type of the document returned by the service (first element of the returned pair);
- type `obj` is the type of the OCaml value returned by the service (second element of the returned pair).

The type of module `Foo.Bar` refines `DynaMoW.Services.Services.SERVICE_TYPE` by providing implementations of the abstract types of the latter in terms of the internal structures.

It is typical in an application to name the service like the file name, and in fact introduce a service `Foo` in the file `Foo.ml`. This results in the frequent need to avoid seeing module names of the form `Foo.Foo`. This is solved by making `Foo` inherit the values `defaults`, `obj`, and `descr` of `Foo.Bar`, but not its types. (This, as a result of a technical limitation in an OCaml module declaration: one can redefine a value, but not a type.) Note that the signature of module `Foo` is, in the end, decided by the programmer (for instance by writing a file `Foo.mli`), and just need to be compatible with `DynaMoW.Services.Services.SERVICE_TYPE`.

One occasionally declares several services in the same file `Foo.ml`. In this situation, module `Foo` inherits only from the lastly defined service, which can then be viewed as a main service: suppose for instance that `Foo.ml` declares a service `Bar` then a service `Foo`, then the main service can be reached by `Foo.descr` and the subservice by `Foo.Bar.descr`.

4. Setting up an application as a (fast) CGI

4.1. Example application using DynaMoW as an FCGI. The distribution of DynaMoW comes with the example directory `doc/example/`, which should serve as a prototype for setting up a new DynaMoW-based application. Here, we base on this example to illustrate the steps of a proper configuration.

4.1.1. *Dependency in DynaMoW and skeleton of application.*

(1) DynaMoW has a few dependencies of which we prefer scripting the installation, rather than basing on packages. These are jsMath and jQuery. The script `install-dependencies.sh` at the top of the DynaMoW distribution will install them at the right place. Setting up an independent application will then be able to copy from those places.

(2) If a fixed version of DynaMoW is to be used, DynaMoW executables can be compiled in view of copying them to the right places in the application. This is simply done by `make`.
(3) For a simple organisation, we propose to split the application into a `src/` directory, where also compilation happens, and a `web/` directory, where only what needs to be visible from the web is stored:
   - copy the files `DynaMoW.cma`, `DynaMoW.cmi`, and `DynaMoW_filter.cma` from the `build/` subdirectory of `dynamow` to the application `src/`;
   - also copy the interface and object files for the symbolic plugins to be used, that is `cas/Maple.cmi` and `cas/Maple.cmo` for Maple, etc.;
   - copy the files `dynamow.css`, `dynamow.js`, `jquery-1.4.2.min.js`, and `loading.gif` from the `web/` subdirectory of `dynamow` to the application `web/`;
   - also make a copy of the subdirectory `jsmath/jsMath/` of `dynamow` as the application `web/jsmath/` (mind the capitalisation change);
   - if Maple is to be used, also copy the file `maple/DynaMoW.mla` of `dynamow` as `maple/DynaMoW.mla` (create the needed directory).

4.1.2. *Structuring the source.* We suggest structuring the source with one service per OCaml file (with consistent names: service `Foo` in file `Foo.ml`, etc.), and to have a separate OCaml file for the application as a fast CGI: in the example application, file `src/Trigo.ml` declares the main service `Trigo` of the application while file `src/Expand.ml` declares an auxiliary service `Expand`, and file `src/Example_fcgi.ml` contains the fast CGI.

Continuing with the example, this main file juste contains:

```
let _ =
  DynaMoW.FCGI.start ~default_service:(Trigo.descr () None) ()
```

which designates service `Trigo` as the default service for the application (with no mandatory argument and no optional argument). After compilation, the application executable is make available to the web server as `web/example`, and is meant to be available by the URL `http://.../example`. When the application starts, it will look for a configuration read from a file named `dynamow.conf`, again under `web/` (the working directory of the fast CGI).

When writing an application, one will very often appeal to exports of the module `DynaMoW.Services.Content`. It is also likely that a given application uses a fixed CAS plugin, and fixed symbolic types. This is why, when an application growth, we suggest having a file `preamble.ml` similar to:

```
use_cas Maple
type_symb maple = unit Maple.t DynaMoW.symb
module DC = DynaMoW.Services.Content
let ( @@@ ) = DC.(@@@)
let ( @:@ ) = DC.(@:@)
```

together with a file `preamble.mli` similar to:

```
type maple = unit Maple.t DynaMoW.symb
```

and to include them at the beginning of your `.ml` and `.mli` files, via `INCLUDE "preamble.ml"` and `INCLUDE "preamble.mli"`, respectively. (Note the use of `type_symb` in the `.ml` file as opposed to the use of `type` in the `.mli` file.)

The example application also contains a toplevel, `example_top`. Having one for your application makes debugging easier. We suggest using it in combination with `rlwrap`.

4.1.3. *Configuring the web server.* Configuring the web server can be delicate, and many factors can adversely affect your debugging, starting with cache. For the benefit of the user and without implying that Apache2 is better for a DynaMoW application than another web server, we distribute the template of configuration file under doc/example/apache2-example.conf.

```
<VirtualHost *:80>
  Servername $SERVER_NAME
  ServerAdmin $YOUR_EMAIL
  ErrorLog /var/log/apache2/dmw_example_error.log
  CustomLog /var/log/apache2/dmw_example_access.log combined
  DocumentRoot /var/www
  Alias /app $ROOT/web
  Appclass $ROOT/web/$APPL -initial-env PATH=/usr/local/bin:/usr/bin:/bin
</VirtualHost>
```

Here, $ROOT is the absolute path to your application directory (of the form .../doc/example for the example application), and $APPL is the name of the standalone application that runs as a fast CGI (example in the example). Warning: $ROOT/web/$APPL should contain no dereferencement through a symbolic link. Additionally, $SERVER_NAME and $YOUR_EMAIL have to be set to reasonable values.

A speed-up in the server responses can be achieved by adding the option -processes 5 (or similar) to the Appclass line.

4.2. Possible forms of query strings. To be written: Query strings can have several possible syntaxes, currently:

- service=...
- metadata=...
- svgpng=...
- svg=...
- latex2png=...

4.3. More about MIME configuration. To be written.

4.4. Default page. To be written.

4.5. Using CSS and ad-hoc web pages. To be written.

4.6. Security. To prevent simple attacks on the server, parameter passing between services is secured by authentication using a message authentication code (MAC). As the MAC takes the service into account, this allows prevention of injection of symbolic code obtained from one service to another.

As a side effect, changing the (secret) key used to compute MACs from one release of an application to the next avoids parameters sets that would be memorized in a user's bookmarks to be used with the new release.

This security measure is currently toggled at compilation time (see PACK_VARIANT in §4.7.1), but this may change in the future. The secret key is set by the configuration variable mac.key (see §4.7.2).

Whether a given DynaMoW application uses authentication by MAC can be seen from the presence of a field mac=... in URLs.

4.7. Advanced configuration. DynaMoW has two configuration mechanisms: compilation options can be set in file config.mk, while the run-time behavior can be controlled through dynamow.conf.

4.7.1. *Compilation.* During compilation of DynaMoW, the file `config.mk` in the sources root directory is read by `make`. The file `config.mk.doc` contains the default values of the configuration options, which are the following:

`MAPLEAPP`:
: The Maple executable to use, if `maple` is not in the `$PATH`.

`COMPONENTS`:
: The list of DynaMoW's components to be compiled. Valid value are: `cas`, `services`, and `fcgi`. For now, this allows one to get just the toplevel functionality of DynaMoW without depending on the ocamlnet library, by only choosing the `cas` component. In the future, there might be components with overlapping functionality, like a standalone web server alongside the FastCGI module.

`PACK_VARIANT`:
: If set to `cryptokit`, query string containing service arguments are augmented by a MAC for authentication. This adds a depency to the cryptokit library. If set to the default value `simple`, the query strings are not authenticated.

4.7.2. *Runtime.* What follows describes the format of the run-time configuration file for DynaMoW. It should be placed in `dynamow.conf` in the program's working directory. (Initial white spaces are not taken into account in variable definitions; simple spaces as well as double quotes are mandatory around the equal sign and value, respectively.)

`core.debug`:
: Enables debug mode if set to `"on"`. Currently, the debug mode includes the display of a backtrace in case of an error and all the interaction with the CAS.

`core.log_file`:
: A path of a file that must be writable by the web server, where debug information is logged. By default, output goes do stdout.

`html.css_file`:
: (Possibly relative) Path to a css file for the generated web pages. Defaults to `"dynamow.css"`.

`html.jquery`:
: URI of a jQuery distribution. Defaults to a local copy, `"jquery-1.4.2.min.js"`. Another possible good choice is `"http://code.jquery.com/jquery-1.4.2.min.js"`.

`mac.key`:
: Key used to authenticate query strings by a message authentication code (MAC). See §4.6 for details.

`maple`:
: Whole section is optional, but required if the Maple plugin is to be used.

`maple.path`:
: Path to the Maple executable. Defaults to `"maple"` (in that case maple should by in your `$PATH`).

`maple.opts`:
: Options of the Maple executable. Defaults to `"-qt -e0"`.

`maple.libs`:
: Additional Maple libraries. Defaults to `""`. Should contain a sequence

of library options, like `"-b ~/maple"`. The Maple core library and the Maple/DynaMoW glue is also always loaded at the end (by the option `"-B -b ../maple"`).

`server.commit_hash`:
Defaults to `"git rev-list --max-count=1 --oneline HEAD"`. DynaMoW also provides a script, available by setting the option to `"../dynamow/bin/commit_hash"`.

`server.cache_max_age`:
Defaults to `"7_257_600"`. Three months. Negative value means no cache.

`server.cookie_max_age`:
Defaults to `"604_800"`. A week.

5. Exceptions raised by DynaMoW at run time

The list and semantics of exceptions are not fully stable yet, but, on an informative level, we can state that DynaMoW raises:

- an exception `Error.DynaMoW_error msg` for an exception produced out of the CAS and intended to be trapped by the application;
- an exception `Error.CAS_error msg` for an exception produced in a CAS and intended to be trapped by the application;
- `failwith` and `invalid_arg` exceptions for debugging purposes. Such exceptions should not appear with stable application code, and thus, most probably reveal a bug in the application;
- `DynaMoW.Invalid_type` and `DynaMoW.Out_of_range` exceptions, already described, when values produced by external CAS execution cannot be handled properly by DynaMoW.

A weakness in the current DynaMoW implementation makes it impossible for the application to catch the first two exception types.

6. Troubleshooting

This section gathers interpretation of and advice about puzzling behaviours of the web server and compile-time errors.

- **A service displays `__dynamow_`*nnn***
  Variables `__dynamow_`*nnn* are used internally by DynaMoW (in fact, by the Maple plugin) to refer to values that have already been computed. If you see that on a web page, it means that DynaMoW lost track of some of your symbolic values, most probably when passing it from a service to another (thus, from one Maple instance to another). Beside an error in the code of your Maple quotations, it can be that `type` was used to declare a symbolic type when `type_symb` should have been used.
- **Compile-time error:**
  `Error: Parse error: illegal begin of top_phrase`
  A space is missing in the construct `<<$(`: type `<< $(` or `<:symb<$(` instead (see §2.2.1).

Appendix A. Output from the service declaration of `LogIntegral` in §3.1.2

```
type maple = unit Maple.t DynaMoW.symb
let_service LogIntegral (n : int) : string * maple =
  let res = << int(log(x)^$(int:n), x) >> in
  (<:string< sprintf("%a", $(res)) >>, res)

module SERVICE_LogIntegral__ :
  sig
    type required = int
    type optional = unit
    type doc = string
    type doc_type = [ `Binary ]
    type obj = unit maple
    val name : string
    val service_type : string
    val param_names : string list * 'a list
    val defaults : unit
    val title : 'a -> 'b DynaMoW.Services.Content.entities
    val style : DynaMoW.Services.Content.css
    val extra : 'a -> [> `Binary ]
    val main :
      required * optional -> string * 'a DynaMoW_CAS__.t DynaMoW.symb
    val to_string : 'a -> 'b -> 'b
    val string_list_of_args : int -> unit -> string list
    val required_of_string_list : string list -> int
    val optional_of_string_list : 'a list -> unit
    val reset_cas : unit -> unit
  end
module LogIntegral :
  sig
    type required = SERVICE_LogIntegral__.required
    type optional = SERVICE_LogIntegral__.optional
    type doc_type = SERVICE_LogIntegral__.doc_type
    type obj = SERVICE_LogIntegral__.obj
    val defaults : optional
    val obj : required * optional -> obj
    val descr :
      required -> optional option -> doc_type DynaMoW.Services.service_descr
  end
module type LogIntegral_TYPE__ =
  sig
    val defaults : LogIntegral.optional
    val obj : LogIntegral.required * LogIntegral.optional -> LogIntegral.obj
    val descr :
      LogIntegral.required ->
      LogIntegral.optional option ->
      LogIntegral.doc_type DynaMoW.Services.service_descr
  end
val defaults : LogIntegral.optional = ()
val obj : LogIntegral.required * LogIntegral.optional -> LogIntegral.obj =
```

```
  <fun>
val descr :
  LogIntegral.required ->
  LogIntegral.optional option ->
  LogIntegral.doc_type DynaMoW.Services.service_descr = <fun>
```