Multilingual Sage*

Jordi Saludes[1], Sebastian Xambó[2]

[1] Sistemes Avançats de Control, Universitat Politècnica de Catalunya, Edifici GAIA, Pg. Ernest Lluch/Rambla Sant Nebridi, 08222 Terrassa, Spain.

[2] Matemàtica Aplicada II, Universitat Politècnica de Catalunya, Edifici OMEGA, c/ Jordi Girona 1-3, 08034 Barcelona, Spain.

E-mail: `{jordi,sebastia}@upc.edu`

Abstract

The aim of this paper is to review the Mathematics Grammar Library (MGL) approach to multilingual mathematics and to present `gfsage`, a prototype of a multilingual interface to Sage.

2000 Mathematics Subject Classification. **68N18.** 68T30, 68T50, 68W30.
Keywords. Multilingual Machine Translation, Mathematics Grammar Library, Multilingual Interfaces to Computer Algebra Systems, SAGE..

Introduction

The problem of multilingual machine translation (MLMT) of mathematical texts is, in all its generality, very hard as seen from a present day perspective (see the detailed analysis by Aarne Ranta in [8]). Even for the much narrower context of MLMT of school word problems, the difficulties are still very high and not fully solvable yet (cf. [11, 12]).

A special case of MLMT that in principle is fully sovable with current techniques is when one of the languages, say S, is the language generated by the input/output grammar of a computational system, and the other language, say X, is a natural language. In this case the problem is to translate commands expressed in X into input commands expressed in S and conversely, to translate output results phrased in S into sentences in X. In symbols, we want to manage the translation $X_i \to S_i$ of the input segment X_i of X to the input syntax of S_i of S and the translation $S_o \to X_o$ of the output syntax S_o of S to the output segment X_o X.

In this paper we focus on the case in which S is Sage [2, 1] and X is any language of a list of natural languages that we will specify in a later section. Actually we can assume, as we will do, that X stands for a fixed language of that list, as the translation of the involved sentences between the languages of the list is well understood by current means. In fact, for simplicity, we will take X to be English (henceforth E).

The organization of the material is as follows. In Section 1 we outline in rather informal terms the general approach to MLMT that we will follow, with special emphasis on the particular problems posed by interfaces to computing systems. Then, in Section 2, we recall the main features of the Grammatical Framework (GF), the programming paradigm on which MOLTO (and other previous

* The research leading to the results presented here has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-247914 (MOLTO project: http://www.molto-project.eu/).

and current projects) is based. Our approach to MLMT of mathematical text is presented in Section 3. The main component is a GF library that we call Mathematical Grammar Libray (MGL). With these ingredients, we can discuss in more detail the problem of producing natural language interfaces to computer algebra systems (CAS) and, more specifically, to Sage (Section 4). In this case, we have developed a prototype interface, `gfsage`, based on MGL, and we devote Section 5 to outline its main features. In Section 6 we include some examples of how the interface works. Finally we summarize some conclusions, and hint at future work, in Section 7.

1 Grammar based MLMT

There are two main approaches to MLMT translation. One is statistical machine translation (SMT) and the other is grammar (or rule) based translation (GMT). We refer to the recent treatises [4] and [7], respectively, for excellent presentations of these two approaches. They are complementary, in the sense that GMT aims at precision, at the expense of coverage, while SMT has a broad coverage at the expense of precision. Incidentally, one of the strong points of the MOLTO project is to advance in the cooperation between these two approaches and the establishing of criteria for optimal compromise, when needed, between them.

Here, due to the nature of the concerns in this paper (natural language interfaces to CAS), we will only consider the GMT approach. This is not unreasonable, as the total precision requirement is at the same time the basis for interfaces driven by a predictive parsing mechanism.

The core of GMT is an abstract *semantic* syntax U, often called an *interlingua*, together with maps $A : E \to U$ (*parser*) and $L : U \to E$ (*linearizer*). The parser extracts the meaning of E-sentences in a language independent form (U-sentences) and the linearizer renders abstract meanings as E sentences.

In the case of wishing natural language interface to a computational system S, the concepts involved and the problems to be solved are summarized by the following scheme:

$$\begin{tikzcd}
\boxed{E_i} \arrow[r, "A_i"] & \boxed{U_i} \arrow[r, "L_i"] & \boxed{S_i} \arrow[d, bend left, "S"] \\
\boxed{E_o} & \boxed{U_o} \arrow[l, "L_o"'] & \boxed{S_o} \arrow[l, "A_o"']
\end{tikzcd}$$

Elements of a natural language interface to a computational system.

Here S_i and S_o denote the input and output syntaxes of the system S, which may be thought as sets of strings. The right hand side of the diagram means that S receives an input string from S_i and that it returns and output string belonging to S_o. On the other hand, E_i and E_o denote the natural language syntaxes designed to express commands to S and to interpret the results, respectively. The core component of the interface is formed by the abstract grammars U_i and U_o, which we may think as a language independent layer. Indeed, the grammar U_i allows us to parse E_i commands to abstract syntax (the map A_i in the diagram) and to linearize the abstract syntax into S_i syntax (the map L_i in the diagram), which is what S understands. Similarly, the grammar

U_o allows us to parse output strings of S into abstract syntax (the map A_o in the diagram) and to linearize that syntax to strings in E_o.

Thus at one end we have S, with its native peculiarities, and at the other the natural language segments appropriate for expressing commands and interpreting results. The link between the two is the semantic layer represented by U_i and U_o. We go from E_i to S_i in two steps: with the parser A_i we abstract E_i commands into U_i form and with the linearizer we render the U_i expressions into strings that are understood by S. In the other direction, the ouput strings of S are abstracted to U_o form by the parser A_o and then the resulting expressions are linearized into natural language strings in E_o.

At this stage it is also important to be aware that the natural languages E_i and E_o will be a mixture of text and formulas, as the same is true for S_i and S_o. The abstract expressions corresponding to such formulas can be thought in terms of encodings such as LaTeX.

2 The Grammatical Framework

In our view, the best choice to implement the functions A_i, L_i, A_o and L_o is the *Grammatical Framework* (GF) [7, 8]. GF is a programming language that allows writing *abstract grammars* and *concrete grammars*. An abstract grammar generates a set of *trees* (abstract syntax) and a concrete grammar for a given abstract grammar associates a string (concrete syntax) to each abstract tree. One of the main features of GF grammars is that these two aspects work as inverse processes: an abstract grammar actually *parses* strings into trees, which capture the semantics of the application domain, and a concrete grammar *linearizes* trees into strings. The multilingual capacity is achieved, roughly speaking, by writing suitable concrete grammars for a given abstract grammar.

Let us see in some more detail a few basic principles that drive GF that we need in later sections. For a complete reference, see [7].

Abstract syntax

To set up the abstract syntax, we have to introduce *categories* and *functions*. To explain the meaning of these concepts, and how they are used, an example should suffice. If we want to talk about natural numbers, we can introduce a category `Nat` to stand for the type of natural numbers and a category `Prop` to stand for propositions about natural numbers. The GF syntax to declare these categories is as follows:

```
cat Nat, Prop ;
```

Now, according to elementary axiomatic matematics (every natural number is zero or the successor of a natural number), we need to express that 'zero is a natural number' and that 'the successor of a natural number is a natural number'. In GF this is achieved with the following code:

```
fun
   Zero : Nat ;
   Succ : Nat -> Nat ;
```

Here `Nat -> Nat` is the GF syntax, borrowed from functional programming, to express the signature of functions from `Nat` to `Nat`.

The signature of predicates such as 'even number' or 'prime number' is `Nat -> Prop`, as they produce a proposition when applied to a natural number. Similarly, the signatures of the 'sum' and 'product' functions are both `Nat -> Nat -> Nat`, and `Nat -> Nat -> Prop` is the signature of the binary predicate 'less than'. Thus we can write

```
fun
   Even, Prime : Nat -> Prop ;
   Sum, Prod : Nat -> Nat -> Nat ;
   Less : Nat -> Nat -> Prop;
```

Finally, the signatures of the logical operations 'not', 'and' and 'or' can be abstracted as follows:

```
fun
   Not : Prop -> Prop ;
   And, Or : Prop -> Prop -> Prop ;
```

Again, the functional programming syntax `Prop -> Prop -> Prop` expresses the type of funtions that map each pair of propositions about natural numbers to a third proposition about natural numbers. For example, the abstraction of '2 is an even number and 2 is a prime number' is the tree `And (Even 2) (Prime 2)`, if we understand that 2 is the tree `Succ Succ Zero`.[1]

In practical terms, the declarations so far would form the *body* of an *abstract module* that would have the form

```
abstract Arith = {<body>}
```

where `Arith` is the name of the module.

Concrete syntax

To any abstract syntax we can associate any number of concrete syntaxes. A concrete syntax associated to an abstract syntax is defined by declarations `lincat`, one for each category C, and `lin`, one for each function f. If C is a category, and T a valid linearization type (usually a linguistic construct), `lincat C=T` means that the linearization type of C is T. Similarly, if f is a function and t a valid function linearization type (usually a syntactic constructor), then `lin f=t` declares that the linearization type of f is t. An example will suffice to see how these declarations work. The listing

```
lincat
   Nat, Prop = Str
```

declares that we will linearize Nat and Prop as Str (the built-in type of strings). As for the linearization of functions, next listings show a few examples. The listing

```
lin
   Zero = "zero" ;
   Succ n = "the successor of" ++ n ;
   Sum n m = "the sum of" ++ n ++ "and" ++ m ;
   Less n m = n ++ "is less than" ++ m;
```

shows verbal linearizations, whereas

[1] In practice we have to deal also with the fact that we say '2 is an even prime number' instead of '2 is an even number and 2 is a prime number'. We refer to [8] for indications of how to address these kind of parsing difficulties.

```
lin
   Zero = "0" ;
   Succ n = n ++ "+" ++ "1" ;
   Sum n m = n ++ "+" ++ m ;
   Less n m = n ++ "<" ++ m;
```

provides formula linearizations. The operator `++` means token concatenation (like string concatenation, but with a space to separate tokens). In the verbal linearization, `Succ Zero` becomes "the successor of zero" and in the formula linearization, "$0 + 1$".

3 Mathematics Grammar Library

Our approach to MLMT of mathematical text is based on the development of a GF application that we call Mathematical Grammar Libray (MGL).[2] This library was presented in [11]. Here we update the background presentation contained in that paper, which we follow closely, but refer to [3] for the interactive demo application `MathBar`.

As in any application coded in GF, we need to specify what categories will be used. In the case of MGL, the most relevant categories are `Value` O and `Variable` O, where O is a `Number`, a `Function`, a `Set` or a `Tensor` (for vectors or matrices). The present version of the library implements these categories by defining a fixed category for each pair in

$$\{\texttt{Value}, \texttt{Variable}\} \times \{\texttt{Number}, \texttt{Function}, \texttt{Set}, \texttt{Tensor}\}.$$

The names of these categories are `VarNum`, `VarFun`, `VarSet`, `VarTen`, `ValNum`, `ValFun`, `ValSet` and `ValTen`. For example, `VarNum` and `ValSet` are the names of the categories `Variable Number` and `Value Set`, respectively. Declarations such as `x : VarNum` and `s : ValSet` mean that `x` is a numeric variable and `s` is a set, like, say, "the domain of the natural logarithm". The distinction between variables and values is useful for type-checking productions like lambda abstractions that require a variable as the first argument. Let us also say that variables can be promoted to values when needed.

Finally, let us remark that the library includes other categories to represent propositions, geometric constructions and indexes.

The library is organised as a matrix. The columns of the matrix correspond to natural languages, which at present are Bulgarian, Catalan, English, Finnish, French, German, Hindi, Italian, Polish, Romanian, Russian, Spanish, Swedish and Urdu. To this list we have added LaTeX, a programming language for typesetting mathematical texts, and the natural language interface to Sage that will be presented in Section 5

The rows of the matrix correspond to three layers, ordered in increasing complexity:

Ground. It deals with literals, indices and variables.

OpenMath. It is modelled after the OpenMath[3] CD's, in the sense that in this layer there is an MGL module for each CD.

[2] The living end of the library is publicly available using subversion as
`svn co svn://molto-project.eu/mgl`.
A stable version can be found at
`svn co svn://molto-project.eu/tags/D6.1`.

[3] A *de facto* standard for mathematical semantics, and usually abbreviated as OM. It is "an extensible standard

Operations. This layer takes care of simple mathematical exercises. These appear in drilling materials and usually begin with directives such as 'Compute', 'Find', 'Prove' or 'Give an example of'.

The following tree is an example of what can be expressed in the *OpenMath* layer:

```
mkProp
(lt_num
  (abs (plus (BaseValNum (Var2Num x) (Var2Num y))))
  (plus (BaseValNum (abs (Var2Num x)) (abs (Var2Num y)))))
```

When linearized, say with the Spanish concrete grammar, it yields

El valor absoluto de la suma de x e y es menor que la suma del valor absoluto de x y del valor absoluto de y.

Similarly, the tree

```
DoSelectFromN
  (Var2Num y)
  (domain (inverse tanh))
  (mkProp
    (gt_num
    (At cosh (Var2Num y))
     pi))
```

gives, when linearized with the English concrete grammar:

Select y from the domain of the inverse of the hyperbolic tangent such that the hyperbolic cosine of y is greater than pi.

Remark. The CD's that appear in the OM layer are those that were considered useful for the WebALT project (http://www.webalt.net/):

- arith1, arith2, complex1, integer1, integer2, logic1, nums1, quant1, relations1, rounding1;
- calculus1, fns1, fns2, interval1, limit1, transc1, veccalc1;
- linalg1, linalg2;
- minmax1, plangeo1, s_data1, set1, setname1.

These CD's suffice to express arithmetic operations, basic relation statements, and calculus of one variable (including derivatives and integrals).

for representing the semantics of mathematical objects, allowing them to be exchanged between computer programs, stored in databases, or published on the worldwide web" (see [6]). It is structured in *Content Dictionaries* (CD's), each of which defines a collection of mathematical objects.

4 Multilingual interfaces to Computer Algebra Systems

According to the scheme introduced in Section 1, and taking into account the discussions in Section 2, in order to have a natural languge interface to a CAS system S, $E \leftrightarrow S$, we need abstract and concrete grammars for E_i, E_o, S_i and S_o. All these grammars depend on S. This is clear for S_i and S_o, but it is also true for E (or X in general), as the minimum natural language segment needed to feed the interface frontend is determined by S.

At this point we see that the MGL presented in Section 3 is a sort of grammatical platform that can be used to approach this problem. So in principle we could produce multilingual interfaces to existing CAS systems with manageable effort. When it comes to choose a first system to try these ideas, Sage distinctly stands out. Aimed at "creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab", Sage is a sophisticated computational environment produced by an on-going collective endeavour led by William Stein (See [5, 2] for a description of the system and its functionalities). The fact that it can be accessed via Internet, makes it a winner on several counts, including its potential role in the teaching/learning of mathematics up to the university level.

5 gfsage

This is a recently developed MGL-based multilingual interface prototype to Sage. In other words, it enables to express Sage commands in natural language and get the results phrased likewise. This section elaborates on the outline presented in [11, 12].

The tool starts a Sage session in the background (as described in Simple Sage Server API, [9]), reads the pgf grammar file[4] and translates the queries from the chosen natural language to the concrete grammar for Sage. This is passed to the Sage server for evaluation and the server replies with a `done` or a `computing` message. In this case the program waits for completion of the computation and then writes the answer.

From the GF side, what is sent to Sage is always in the category `Command`. What is returned by Sage is in the category `Answer`. There are 3 kinds of Commands:

- Asking for a computation.
  Compute: `Kind -> Value Kind -> Command`.

  Sage gives back a `ReturnBlock` with the cell number and the answer (a string). We could now construct a short `Answer` by using:

  - Simple: `k` $\in$ `Kind -> Value k -> Answer`
    ("it is 5"), or
  - Feedback: `k` $\in$ `Kind -> Value k -> Value k -> Answer`
    ("the factorial of 3 is 6"), that combines the question (the first Value k) with the Sage answer.

- Assuming propositions.
  Assume: `Prop -> Command`.

[4] pgf stands for portable grammar format, which is a low-level binary format obtained by compiling GF grammars, a facility included in the GF shell. For details, we refer to [7], Chapter 7.

Sage silently accepts the command by returning an `EmptyBlock` (with cell number) but we want it to be more assertive, so we reinject the `Prop` into Assumed: `Prop -> Answer`
("I assume that x is greater than 2")

- Binding Values to Variables.

  Assign: `k` $\in$ `Kind -> Var k -> Value k -> Command`
  ("assign 2 to x").

  We expect Sage to return an `EmptyBlock` followed by

  Assigned: `k` $\in$ `Kind -> Var k -> Value k -> Answer`
  ("2 is now assigned to x").

Let us conclude this section with a brief summary of the main current features of `gfsage`:

- With `gfsage` one can express arithmetic operations, basic relations and calculus of real functions of a real variable (derivatives and integrals, both definite and indefinite). This limited coverage is already a suitable baseline for applications such as those outlined in [12].
- The `gfsage` input is driven by a predictive parser. This is a user-friendly feature that guarantees that the input sentences follow the strict rules of the MGL grammar.
- Scalability, maintainability and multilingual quality are among the strong points of `gfsage`. They are consquences of the GF and MGL architecture. The inclusion of a new language, for example, is straightfoward as soon as the corresponding GF linguistic components (resource grammar and lexicon) are available.
- The motivation for `gfsage` has been not only its intrinsic interest in the context of quality multilingual machine translation, but also the wish to produce a useful system for the regular Sage users, particularly those that are not familiar with a CAS. Since it is based on the cooperation between GFa and a CAS, mediated by the MGL, any improvement in GF may be reflected in an improvement of `gfsage`.
- Another interesting aspect of `gfsage` is that it can suggest Sage packages that are not yet available. See [10] for an example meant to express domains and ranges of real functions of a real variable.
- The current version of `gfsage` is not very easy to install. The complications stem from the strong differences between the programming context of Sage (Fortran, C, Python) and that of GF (Haskell). This will improve in the near future due to the fact that GF is in the process of enabling to interface other languages at runtime (for the moment this has been completed for C).

6 Examples

Here are some illustrations:

```
compute the sum of 1, 2, 3, 4 and 5.
   15
```

```
compute the summation of x when x ranges from 1 to 100.
   5050

compute the integral of the cosine on the open interval
from 0 to the quotient of pi and 2.
   1

compute the integral of the function mapping x
to the square root of x on the closed interval from 1 to 2.
   4/3*sqrt(2) - 2/3

compute the product of the octal number 12
and the binary number 100.
   40

compute the imaginary part of the derivative
of the exponential at pi .
   0

let A be the open interval by the right from 0 to 4 .
let B be the closed interval from 3 to 5 .
let C be the intersection of A and B .
let D be the open interval from 2 to 4 .
unite C with D .
   ] 2 , 4 [
   it is the open interval from 2 to 4 .
compute the intersection of A and D .
   ] 2 , 4 [
   it is the open interval from 2 to 4 .

compute the domain of the inverse function
of the hyperbolic tangent .
   ] -1 , 1 [
   it is the open interval from minus 1 to 1 .

let f be the arctangent.
compute the domain of f.
   ] -Infinity , +Infinity [
   it is the open interval from minus infinity
   to infinity .

compute the left-composition of the sine
and the cosine.
   X |--> sin(cos(X))
   it is the function mapping X to the sine
   of the cosine of X .

let A be the open interval by the left from 0 to 4 .
let B be the closed interval from 4 to 5 .
compute the intersection of A and B .
   {4}
   it is the set with only element 4 .
```

```
let A be the open interval by the right from 0 to 4.
let B be the closed interval from 3 to 5.
let C be the intersection of A and B.
let D be the union of A and B.
compute the difference of D and C.
   { [ 4 , 5 ] , [ 0 , 3 [ }
   it is the union of the closed interval from 4 to 5
   and the closed interval by the left from 0 to 3 .
let D be the set with elements 1,2,3 and 4.
compute the intersection of A and B.
   [ 3 , 4 [
   it is the closed interval by the left from 3 to 4 .
unite it with D.
   { [ 3 , 4 ] , {1 , 2} }
   it is the union of the closed interval from 3 to 4
   and the set with elements 1 and 2 .
compute the union of A and D.
   { [ 0 , 4 ] , {2} }
   it is the union of the open interval by the left
   from 0 to 4 and the set with only element 2 .
compute the difference of D and the set with only
element 4.
   {1 , 2 , 3}
   it is the set with elements 1 , 2 and 3 .

let A be the closed interval from 0 to 1.
let B be the open interval from 2 to 3.
let C be the open interval from 3.1 to pi.
let D be the set with elements 0, 2, pi, 4 and 5.
compute the union of A, B, C and D.
   {[0,1], ]3.10000000000000, pi], [2,3[, {4, 5}}
   it is the union of the closed interval from 0 to 1 ,
   the open interval by the left from 3.1 to pi ,
   the closed interval by the left from 2 to 3 and
   the set with elements 4 and 5.
```

For more information, including news about how `gfsage` is evolving, search 'gfsage' at the MOLTO web page

http://www.molto-project.eu/.

There you can find, for example, interesting details about how to enhance `gfsage` with multilingual voice.

Other systems worth mentioning include *WolframAlpha* [13]. Powered by *Mathematica*, it covers grounds that go far beyond computing, but it is monolingual and fails to understand, as far as the authors can tell, pure mathematical complex queries.

7 Conclusions and further work

In this paper we have explored applications in the realm of Mathematics of the GF approach to machine multilingual environments. In particular, we have analysed the problem of multilingual interfaces to computational mathematics systems. As a platform for these kind of applications, we have presented the MGL. Finally we have presented the prototype interface `gfsage`.

Here are a few lines for future work:

- To enrich `gfsage` with a GUI that allows to enter text and formulas by means of icon palettes. The text icons, like those featured in the MathBar prototype [3], support predictive writing and ensure that the resulting texts are grammatically correct. The formula icons, like those implemented in many systems, ensure that functions are given the appropriate number and kind of arguments. Together, the enriched interface would guarantee that the resulting multilingual texts are syntactically sound, both linguistically and mathematically.

- To continue with the development of MMMA (Multilingual Mechanical Mathematics Assistants) outlined in [12],[5] In this case, in addition to the MGL and `gfsage`, the main component to be developed is an interface to a suitable TP (Theorem Prover). In this regard, a worthwhile project would be to develop a TP-Sage. Along these lines, a good start will be interfacing with the Geogebra TP for Geometry.

Acknowledgements

We are thankful to the referees for a number of suggestions that have definitely improved the final version.

Bibliography

[1] F. Botana, J. Escribano and M. A. Abánades, *Sage: una aplicación libre para matemáticas*, SUMA 67 (2011), 41–46.

[2] Sage developing team, *Sage Tutorial, release 4.7.2.* Sagemath.org, 2011.

[3] T. Hallgren and J. Saludes, *Math Bar Online*, 2010. `http://cloud.grammaticalframework.org/minibar/minibar.html`.

[4] P. Koehn, *Statistical machine translation*, Cambridge University Press, 2010.

[5] T. Kosan, *Sage for Newbies*, Sagemath.org, 2007.

[6] The OpenMath Society: `http://www.openmath.org`.

[7] A. Ranta, *Grammatical Framework: Programming with Multilingual Grammars*, CSLI Publications, Stanford, 2011.

[8] A. Ranta, *Translating between language and logic: what is easy and what is difficult*, In Proceedings of the 23rd international conference on Automated deduction, CADE'11, pages 5–25. Springer-Verlag, 2011.

[9] Sage developing team, Simple Sage Server API, Consulted Jan 15, 2012.

[5] See also `http://www-ma2.upc.es/sxd/Talks.html`, where you can find the slides of the talk.

[10] J. Saludes and A. Ribó, *Reals sets consisting of intervals and isolated points, supporting integration*, http://trac.sagemath.org/sage_trac/ticket/13125.

[11] J. Saludes and S. Xambó, *The GF mathematics library*, EPTCS, 79 (2012), 102–110. Proceedings THedu'11.

[12] J. Saludes and S. Xambó, *Toward multilingual mechanized mathematics assistants*, In J. R. Sendra and C. Villarino, editors, Resúmenes del XIII Encuentro de Álgebra Computacional y Aplicaciones (EACA 2012), number 09 in Obras Colectivas Ciencias, pages 163–166. Universidad de Alcalá, June 2012.

[13] WolframAlpha, *WolframAlpha: Computational knowledge engine.* http://www.wolframalpha.com.