

JAVA Phrasebooks for Computer Algebra and Automated Deduction

Olga Caprotti

Arjeh M. Cohen

Manfred Riem

Eindhoven University of Technology
Faculteit Wiskunde en Informatica
Postbus 513
5600 MB Eindhoven
{olga,amc,mriem}@win.tue.nl*

Abstract

We discuss the developments within the *OpenMath* framework regarding programs that make it possible for a software package to interact with other packages or agents, namely *Phrasebooks*. Recently, several implementations of *Phrasebooks* have come about; we shall describe some of them. Most of the software is freely available, so, by downloading it and inspecting implemented examples, builders of software packages can pick up the examples and provide the mathematical community with further computational servers that can easily be interfaced.

1 Introduction

The integration of computer algebra packages and proof checkers in an interactive computer environment is achieved by means of *OpenMath Phrasebooks*. Their task is to translate the *OpenMath* object, as understood by means of Content Dictionaries (CDS), to the corresponding internal representation used by the specific software package, to monitor a specific action upon receiving the object, and to convert the internally obtained output back into an *OpenMath* object.

Several *Phrasebooks* are under development as part of the *OpenMath* Esprit Consortium project. Prototype versions of *Phrasebooks* for the computer algebra packages AXIOM, GAP, Mathematica, Maple, and the proof checker COQ are already available.

In this paper, we shall go somewhat deeper into the functionality and design of *Phrasebooks*, with the intent of making clear to those who wish to interface software, what needs to be done in order to succeed within the *OpenMath* framework.

2 OpenMath Compliant Phrasebooks

There are several aspects to *Phrasebooks* that we shall discuss on an ‘imaginary journey’ of an object A through a software package. Assume that we have a *Phrasebook* for the package. There will be several components to this *Phrasebook*. To begin, there is a *list of CDS* that it recognises. Furthermore, it has an encoder from *OpenMath* objects to an internal representation, and vice versa (decoder). We refer to this part of the *Phrasebook* as *codec*. Next, there is an *interpretation* of what is done to the object with the package. The interpretation is a function of the control information and the received *OpenMath* objects.

*Supported by the OpenMath Esprit Project 24.696.

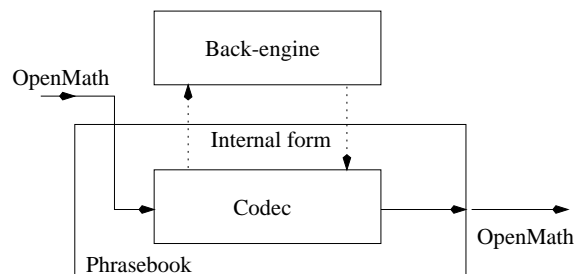


Figure 1: General structure of a phrasebook

Control information conveys the task that the *Phrasebook* performs upon receipt of the object, e.g., EVAL, SIMPLIFY, PROVE, SOLVE, PRINT. When no control information is available, then this task is implicitly defined by the encoded object. The description of the way incoming objects are handled may have a default description, such as ‘evaluation’ for computer algebra systems and ‘print’ for printing devices. Still, it will be useful for the ‘interpretation’ to be described explicitly especially for *Phrasebooks* that interface to computation engines capable of carrying out a variety of tasks. We do not want to imply here that it should be disclosed how the package works internally, but rather that the type of actions corresponding to certain *OpenMath* objects and control directives should be recorded. Some recent formalised approaches on how to specify mathematical services provided by *OpenMath Phrasebooks* are found in [1]. Finally, the *Phrasebook* takes care of the actual *communication* between the software package and the *OpenMath* computer environment. These four aspects are summarised in Table 1.

Table 1: *Phrasebook* functions

task	accompanying information
codec	list of CDS
communication	interpretation

The codec and communication already appear in the *OpenMath* standard, as well as the notion of CDS involved in a *Phrasebook*. So let us focus here on the interpretation aspect.

As a first example, consider the *OpenMath* object A corresponding to $1 + 2$,

$$\text{application}(\text{arith1:plus}, 1, 2) \quad (1)$$

where `arith1:plus` is the symbol `plus` defined in the CD `arith1` and `1` and `2` represent basic objects (arbitrary length integers).

The *OpenMath* to \TeX *Phrasebook* [7] will encode this expression as $\$1+2\$$, the way you expect to typeset it in \TeX source code. This is to be contrasted with a package like *Maple* or *Mathematica*; their *Phrasebook*'s codecs will translate A into $1 + 2$, but the computer algebra system will rewrite it to 3 (performing the implied “evaluate” command), and so 3 , decoded into the corresponding *OpenMath* object, will be returned to the environment.

Things might be not so clear even for the relatively well understood ‘evaluation’ instruction implicitly carried by an object fed to a computational engine. In such cases, it is advisable for the *Phrasebook* explicitly to state unexpected behaviour. Consider a slightly more complicated *OpenMath* object, namely the one for

$$\int_0^1 \left(1 - \frac{1}{2 + e^{2\pi i x}}\right) dx.$$

It involves, amongst others, the CDs `arith1` for $e, +, -, *, /$, `~` and `nums1` for π and imaginary i . More precisely, it looks like this:

$$\begin{aligned} A := & \text{application}(\text{calculus:defint}, & (2) \\ & \text{application}(\text{interval:interval_cc}, 0, 1), \\ & \text{binding}(\text{fns1:lambda}, x, \\ & \quad \text{application}(\text{arith1:minus}, 1, \\ & \quad \quad \text{application}(\text{arith1:divide}, 1, \\ & \quad \quad \quad \text{application}(\text{arith1:plus}, 2, \\ & \quad \quad \quad \quad \text{application}(\text{arith1:exp}, \\ & \quad \quad \quad \quad \quad \text{application}(\text{arith1:times}, 2, \\ & \quad \quad \quad \quad \quad \quad \text{nums1:pi}, \text{nums1:i}, x)))))) \end{aligned}$$

As in the first example, we may feed the expression to *Maple* or *Mathematica*, which will both automatically evaluate it. The results are as in the first two lines of Table 2. The role of the *Phrasebook*'s interpretation of the object A becomes explicit in the alternative on third line of the table, where, instead of the arithmetic expression $f := 1 - \frac{1}{2 + e^{2\pi i x}}$, its ‘simplified’ version `simplify(f)` appears. The effect is drastic; it is recorded in the third column of the table.

Table 2: Three interpretations of A of (2)

package	interpretation of A	effect
Mathematica	<code>Int[1-1/(2+Exp[2 Pi I x], x=0..1]</code>	$1/2$
Maple V, r. 4,	<code>int(1-1/(2+Exp(2*Pi*I*x)), x=0..1)</code>	1
Maple V, r. 4,	<code>int(simplify(1-1/(2+Exp(2*Pi*I*x)), x=0..1)</code>	0

It is desirable for *Phrasebooks* to document design features that deviate from the default interpretation. In this case,

if we denote the interpretation function as binary function $[-, _]$ taking control information and an *OpenMath* object and returning an *OpenMath* object, the *Phrasebook* can be described by:

$$[A] = [\text{EVAL}, A]$$

except when $A = \text{application}(\text{arith1:minus}, A_1, A_2)$, in which case

$$[A] = [\text{EVAL}, [\text{SIMPLIFY}, A]].$$

Of course, the exception rule can be extended to other symbols representing operators. The differences in interpretation in this example are very slight here in that all three versions return an *OpenMath* object which is a real number (setting aside the issue that two answers were in fact integers—in general one would expect a real number). Of course, in this example, we were being nitpicky about an error in *Maple*, discovered by Eric Opdam, and removed in later versions of *Maple*. But the point we are trying to make is that the *OpenMath* language by itself does not prescribe *how* the software package should handle an incoming object—and therefore it should be recorded with the *Phrasebook*.

The third example is a more generic one and describes the approach we have taken for a *Phrasebook* interfacing an extension of a backengine. It is often the case with computer algebra users that they program new functionalities thus extending the set of built-in functions of the package. What we describe now is a way in which new functionalities implemented to support the next release of ‘Algebra Interactive’ [6] are exported using *OpenMath*. Here, for the purpose of an interactive book, we have written several algebraic functions that supply human interaction. For example, the function `RootSumPolynomial` takes as input two irreducible polynomials in $\mathbf{Q}(x)$ and returns a polynomial with the property that the sum of any root of one of the two input polynomials with any root of the other polynomial is a root of the output polynomial. These functions are part of a GAP package called `ida.g` extending the set of built-in GAP functions, see Figure 2.

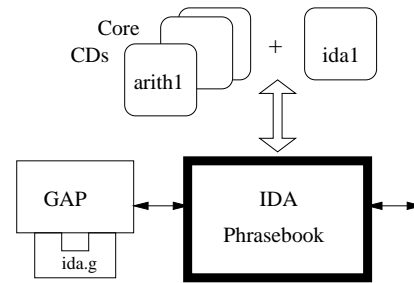


Figure 2: The *OpenMath* *Phrasebook* to the GAP engine for IDA

In order to provide an *OpenMath* interface to this extension of the backengine, we first record the names of all these algebraic functions in a private CD, called `AlgebraInteractive1` (`ida1` for short in Figure 2). (Actually, it may be wiser to build up several CDs than a single one covering all topics, but the choice makes little difference for the sake of the discussion, so we disregard the option

here.) In this way, any computer algebra package will be able to serve as backengine to the interactive book if it has a *Phrasebook* supporting the CD `AlgebraInteractive1`.

There are two possible ways to implement such a *Phrasebook*, depending on whether there are native routines in the backengine for each of the *OpenMath* symbols defined in the CD `AlgebraInteractive1`. If functions like `RootSumPolynomial`, are implemented as procedures in the backengine's language, then the *Phrasebook*'s interpretation in this case requires to carry out this particular procedure by the extended backengine and the codec is simply in charge of producing the proper functional call. Otherwise, when there are no native functions available, the codec might encode the function `RootSumPolynomial` by the body of such a procedure, and then communicate it to the backengine (without extending it). In both cases, (in absence of any additional control information) the *Phrasebook* must carry out evaluation of a function satisfying the properties specified in the CD.

In conclusion, a necessary condition for a *Phrasebook* to be *OpenMath* compliant with respect to a given set of CDs is that it can encode all the *OpenMath* objects whose symbols come from the union of the CDs. In addition, the *Phrasebook* should be able to take care of communication with an *OpenMath* environment. Finally the interpretation of the *OpenMath* objects should be in accordance with the definitions given in the CDs and with the control information it handles. The interpretation reflects whatever the package does to the received objects and, in particular, what it returns. Admittedly, the latter is vague. To a certain extent, that is on purpose, because part of the enchantment of mathematics is that one may receive surprising answers to queries. On the other hand, the flexibility of *OpenMath* also allows us to approach the *Phrasebook*'s interpretation very rigidly, as may be clear from the fact that proof checkers can be integrated into the *OpenMath* environment as well, cf. [4].

3 Java Software

In order to build *Phrasebooks*, it is convenient to use JAVA libraries. Examples of work in this direction are a unique interface based on *OpenMath* to various packages for computer algebra within the CATHODE project [2] and the web interface to Maple by [11].

In our setting, *OpenMath* objects can either be sent and received in JAVA applets or dealt with by JAVA servlets. The latter approach requires server-side programming but avoids having to take care of network security issues. The *OpenMath* objects are encoded in XML. For instance, the *OpenMath* object of (1) is encoded as

```
<OMOBJ>
  <OMA><OMS cd="arith1" name="plus">
    <OMI>1</OMI>
    <OMI>2</OMI>
  </OMA>
</OMOBJ>
```

Best known are the Naomi JAVA Library by the PolyMath Development Group [12] and the one by INRIA [10]. They provide classes representing *OpenMath* objects. In the former one, it is possible to encode and decode arithmetical expressions written in Mathematica or in Maple syntax by using their respective *Phrasebooks*. So here the codec

part of the *Phrasebooks* lives completely outside the software packages to be interfaced. Similar translation features exist for more languages for instance from MathML to *OpenMath* and back, in STARS [3].

Phrasebooks providing an interface to and from *OpenMath* have been built into experimental versions of both AXIOM and GAP, cf. [8, 9]. We however, have adopted the approach of building the full *Phrasebook* outside the software package we want to interface. By use of the Naomi JAVA library, such *Phrasebooks* have already been built for the proof checkers Lego and Coq, for the computer algebra package Mathematica and for the Group Theory package GAP.

3.1 Codecs

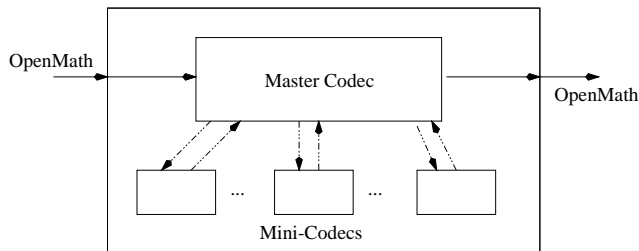


Figure 3: General structure of a codec

An important feature to take care of when building *Phrasebooks* is their extensibility with respect to newly added CDs. To this end, our JAVA codecs have the capability of dispatching part of its encoding and decoding to so-called *mini-codecs*. So the codec either encodes a given *OpenMath* object by creating the encoding by itself, or by sending it off to the instantiated mini-codec. A default solution is needed in case, for a given symbol, no suitable mini-codec is found. The current implementation takes a conservative compliant approach and returns an error upon failure of the encoding.

3.2 Communication

We go into further detail regarding the communication with the software packages Mathematica and GAP. Since the JAVA software we created is freely available¹, our description here might help mathematical software builders, and to some extent even software users, to create their own *OpenMath* interface quite rapidly.

For interfacing GAP, we wrapped a server around the shell interface of GAP. Interfacing with Mathematica was done by using the JLINK, the JAVA link library provided by Mathematica. This is our preferred way of interfacing, because it obviates the need for building a communication with the software package.

If there is no JAVA link, there might be a C callable version of the software package. Usually, with some effort, by wrapping JAVA around the C callable version, a JAVA link can be created. This brings us back to the preferred option—at least theoretically; in practice this solution is definitely not optimal, as control over the software package is limited.

¹Contact the authors if interested.

Since GAP does not allow other programs to operate in its own address space, the above linking options for the package are not feasible. In our solution, written entirely in JAVA, we open pipes from/to the package and conduct the communication by reading from and writing to the pipes. A complication here is that one needs to detect the end of an output message coming from the package. In GAP, a switch is available that marks beginning and ends of inputs and outputs with special characters. This feature enabled us to supply proper communication with GAP for Algebra Interactive. If such a feature does not exist and (the author(s) of the program cannot be persuaded to install it), one may need to do it one-self, perhaps by tinkering on the programming level of the software package.

4 Two Java Phrasebooks

In this section we briefly discuss the design features of the type-checking *Phrasebook* for *Strong OpenMath* objects [5] and of the IDA *Phrasebook* supporting user-interaction in an electronic textbook [6].

4.1 Type-checking Phrasebook

The type-checking *Phrasebook* is an *OpenMath* application that uses the proof assistant COQ to perform type checking on *OpenMath* objects. In particular, when the objects are *Strong OpenMath* objects, then the *Phrasebook* decides if the objects are well typed or not. If they are, then they correspond to mathematical objects with a well-defined meaning.

Let us start by a brief description of the COQ codec. The specification of the COQ codec is exemplified by rules like the following that translate *OpenMath* objects representing lambda calculus terms into COQ syntax.

```

attribution(A, ecc: type t, ...) →  $\hat{A}: \hat{t}$ 
attribution(A, S t, ...) →  $\hat{A}$ 
binding(lc: Lambda, attribution(v, ecc: type t), A) → (v :  $\hat{t}$ )  $\hat{A}$ 
binding(lc: PiType, attribution(v, ecc: type t), u) → [ $\Pi v: \hat{t}$ ]  $\hat{u}$ 
binding(B, attribution(v, ecc: type t), A) → ( $\hat{B} \hat{t} (v: \hat{t}) \hat{A}$ )
application(F, A) → ( $\hat{F} \hat{A}$ )

```

where \hat{A} denotes the recursive application of the rules to the object A . These rules are extended to handle inductive types, written in *OpenMath* using experimental CDs.

A rough description of the COQ *Phrasebook* could say that the default interpretation of an *OpenMath* object is a CHECK command performed by COQ on the decoded object. However the *Phrasebook* does much more than that. Since COQ does not know anything about *OpenMath* objects, and in particular it does not know the *OpenMath* symbols, before being able to type check a generic object, the COQ context has to be updated by the signatures of the *OpenMath* symbols occurring in the object. These signatures are the ones available in *OpenMath* signature files in a type system compatible with that of COQ. A further manipulation is done to the object: all floating point numbers, strings, and bytearrays are replaced by new constants of an appropriate type. This is a safe approach if we use COQ purely as a checker in which new universes corresponding to the *OpenMath* basic objects are declared.

In some cases, the type-checker returns an *OpenMath* object in which some parameter has been instantiated. More precisely, COQ's type inference is used to instantiate the correct type for the arguments. This occurs for overloaded

OpenMath functions like `arith1:plus`. In this case, the formal signature of `arith1:plus` specifies that the type of the arguments is given as first argument; however, we have to address type-checking objects like (1) in which the extra argument is not given. The *Phrasebook* exploits a feature of COQ in which type-checking infers, if possible, missing arguments that are expressed by the placeholder `?`. Hence, instead of type-checking the object A it type-checks a variant of A :

```

application(arith1:plus, ?, 1, 2)

```

and returns the fact that `integer` has to be the domain of `arith1:plus` for this object to be meaningful.

4.2 IDA Phrasebook

The solution given in Section 3.1 for implementing codecs is used for the GAP codec. The encoder takes as input an *OpenMath* object and, using mini-codecs, translates it to the corresponding GAP syntax. For instance, the *OpenMath* object (1) is sent to GAP by using the phrasebook which calls the codec to encode the *OpenMath* object to the GAP syntax:

```

((1)+(2));

```

The reverse direction, in which an expression in GAP is translated to the corresponding *OpenMath* object, is as usual much more involved. The reason for this is that GAP uses the same syntax for expressing different mathematical objects, e.g. a matrix, a list, an encoding of a permutation. When no extra knowledge is available, the GAP decoder interprets the expression as an *OpenMath* string. However, if extra knowledge is available (think of an interactive session in which the expression received from GAP is the result of a computation), then the decoder may use this extra knowledge to return the proper *OpenMath* object in the given context.

In Section 2 we have described how the GAP codec has been extended to deal with symbols coming from a private CD for usage within IDA *Phrasebook*. The IDA codec uses the CD specified in the *OpenMath* object to determine whether it has to handle the input by a mini-codec, or if it is enough to translate it by the general GAP codec, as drawn in Figure 3.

Clearly *OpenMath* is useful when it comes to double checking answers from different packages to the same query, as we have seen in the discussion of the *OpenMath* object A in (2) above. To illustrate that *OpenMath* is also useful in providing a platform for complementary actions to be conducted by different packages, consider the following problem handed out to students of a Linear Algebra class at Eindhoven University of Technology:

Given a prime number p and a positive integer n , determine all conjugacy classes of $n \times n$ matrices with entries in $\text{GF}(p)$, the finite field of order p .

Since the entries can be represented by natural numbers in the range $\{0, \dots, p-1\}$, and arithmetic mod p can be conducted very easily by use of global rewrite rules, many steps of the solution to this problem can be algorithmically carried out in Mathematica. The students were supposed to list all possible minimal polynomials and characteristic polynomials, in order to set up representative matrices (in

rational Jordan normal form) of all classes, and to determine their centralisers (that is to say, their orders) in the general linear group on the vector space $\text{GF}(p)^n$. Finally, the sum of the inverses of the orders of the centralisers should be checked to be

$$p^{n(n+1)/2}/((p^n - 1)(p^{n-1} - 1) \cdots (p - 1)).$$

The only step that is hard to carry out in Mathematica is the determination of (orders of) centralisers, which is a builtin command in GAP. Therefore, an interface between the two would be a natural solution. *OpenMath* can provide it.

5 Conclusion

We have given here an overview of the ongoing work in *OpenMath Phrasebooks* and described several applications. Although this is work in progress, from our experiments, we are confident that *OpenMath* is a good content language for providing mathematical services using computer algebra and automated deduction software.

References

- [1] *Proceedings of CALCULEMUS-2000, 8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2000.
- [2] M. Berth, F.-M. Moser, and A. Triulzi. Implementing Computational Services Based on OpenMath. Submitted. <http://paul.math-inf.uni-greifswald.de/Cathode2/omws/>.
- [3] Stephen Buswell. STARS. Available from Stilo Technologies, April 1999. <http://www.stilo.com>.
- [4] Olga Caprotti and Arjeh Cohen. Connecting proof checkers and computer algebra using OpenMath. In *The 12th International Conference on Theorem Proving in Higher Order Logics*, Nice, France, September 1999.
- [5] Olga Caprotti and Arjeh Cohen. Integrating Computational and Deduction Systems Using OpenMath. In *Proceedings of Calculemus 99*, Trento, July 1999.
- [6] A. M. Cohen, H. Cuypers, and H. Sterk. *Algebra Interactive, interactive course material*. Number ISBN 3-540-65368-6. SV, 1999.
- [7] OpenMath Consortium. T_EX to OpenMath Converter, June 2000. Available at <http://www.nag.co.uk/projects/OpenMath.html>.
- [8] OpenMath Consortium. Axiom interface to openmath. OpenMath ESPRIT Deliverable, 2000.
- [9] OpenMath Consortium. Gap interface to openmath. OpenMath ESPRIT Deliverable, 2000.
- [10] OpenMath Consortium. OpenMath JAVA and C Library, June 2000. Available at <http://www.nag.co.uk/projects/OpenMath.html>.
- [11] Ha Le and Chris Howlett. Client-Server Communication Standards for Mathematical Computation. In *Proceedings of ISSAC'99*, pages 299–306, Vancouver, Canada, 1999. ACM, New York,.
- [12] PolyMath OpenMath Development Team. Java OpenMath Library: Version 0.7c. <http://pdg.cecm.sfu.ca/openmath/>, June 1999.