LEMA: Towards a Language for Reliable Arithmetic

Vincent Lefèvre
INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
vincent.lefevre@ens-lyon.fr

Philippe Théveny
INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
philippe.theveny@ens-lyon.fr

Florent de Dinechin
ENS de Lyon, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
florent.de.dinechin@ens-lyon.fr

Claude-Pierre Jeannerod
INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
claude-pierre.jeannerod@ens-lyon.fr

Christophe Mouilleron
ENS de Lyon, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
christophe.mouilleron@ens-lyon.org

David Pfannholzer
INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
david.pfannholzer@ens-lyon.fr

Nathalie Revol
INRIA, LIP (UMR 5668 CNRS - ENS de Lyon - INRIA - UCBL), Université de Lyon
nathalie.revol@ens-lyon.fr

Abstract

Generating certified and efficient numerical codes requires information ranging from the mathematical level to the representation of numbers. Even though the mathematical semantics can be expressed using the content part of MathML, this language does not encompass the implementation on computers. Indeed various arithmetics may be involved, like floating-point or fixed-point, in fixed precision or arbitrary precision, and current tools do not handle all of these.

Therefore we propose in this paper LEMA (Langage pour les Expressions Mathématiques Annotées), a descriptive language based on MathML with additional expressiveness. LEMA will be used during the automatic generation of certified numerical codes. Such a generation process typically involves several steps, and LEMA would thus act as a glue to represent and store the information at every stage.

First, we specify in the language the characteristics of the arithmetic as described in the IEEE 754 floating-point standard: formats, exceptions, rounding modes. This can be generalized to other arithmetics. Then, we use annotations to attach a specific arithmetic context to an expression tree. Finally, considering the evaluation of the expression in this context allows us to deduce several properties on the result, like being exact or being an exception. Other useful properties include numerical ranges and error bounds.

PLMMS-2010 8^{th} July 2010, Paris.

Categories and Subject Descriptors D2.4 [*Software engineering*]: Software/Program Verification—correctness proofs, reliability, validation

General Terms Design, Languages, Performance, Reliability, Verification

Keywords Annotated mathematical expressions, XML, MathML, Certified numerical properties, Formal proof, Computer arithmetic, Floating-point arithmetic

1. Introduction

A major problem with numerical applications (from libraries to user code) is the control of accuracy with acceptable performance. While some applications are more focused on performance, for other ones, accuracy of the results is crucial. For instance, the requirement can be:

- a guarantee on an error bound specified by the user;
- *correct rounding* to some given target format, that is, the result obtained as if all internal computations were carried out in infinite precision before the final rounding;
- an exact result, when it is known to be exactly representable in some format.

In addition to accuracy, the range of the computed values needs to be considered too, in order to either avoid overflows and underflows, or control what happens if such exceptions occur. Arithmetics for which this kind of properties can be guaranteed are referred to as *reliable arithmetics*.

Even applications focused on performance need meaningful results, which one may want to certify.

Values (either final or internal results) can be expressed in various arithmetics, which can be mixed even within a common application for, say, performance reasons. The most common ones are (see [1] and [24] for comprehensive descriptions):

- integers, either bounded (in a fixed format) or in arbitrary precision;
- floating point in some radix β (in general, $\beta = 2$ or 10) and precision p (fixed or arbitrary): a number x has the form $x = s \cdot m \cdot \beta^e$, where $s = \pm 1$ is the sign, $m = x_0.x_1x_2 \dots x_{p-1}$ (with $0 \leq x_i \leq \beta - 1$) is the significand, and the integer e is the exponent (whose range can be bounded or not);
- fixed point, which is quite similar to integer arithmetic.

But one can also construct derived arithmetics, such as double-double arithmetic [9, 23], where each number x is represented by a pair of double-precision floating-point numbers $(x_{\text{hi}}, x_{\text{lo}})$ and interpreted as $x = x_{\text{hi}} + x_{\text{lo}}$ (exactly).

The variety of architectures or inputs implies the support of some previously mentioned arithmetics, which in turn yields many variants in the implementation of a single algorithm. In order to avoid tedious work for similar solutions, we seek to generate efficient code automatically.

However, speed is not our main concern. We also want to use the same description of inputs and of the chosen arithmetic so as to prove the validity and accuracy of the implementation with formal methods. Indeed, a traditional solution is to validate the programs with hand-written proofs. In the domain of numerical applications, such proofs often require many tedious calculations, and in practice, they often contain errors (e.g., in corner cases), making them globally incorrect. Moreover if the problem slightly changes (due to a different hypothesis, architecture, input...), the whole proof may need to be manually checked or even redone.

To satisfy these needs, we present here a new descriptive language called LEMA.

The outline of this paper is as follows. We start in Section 2 by presenting several established software environments. We show there why they do not allow us to achieve directly the goals mentioned above, and how the single language LEMA could act as a glue between some of them. After describing the LEMA language requirements in Section 3, we argue our decision to use XML as a basis for LEMA in Section 4. Then Section 5 illustrates with the help of examples the use we make of the tree nature of XML and its extensibility. In particular, we show how mathematical expressions are annotated with their evaluation in a given floating-point context and with associated proofs.

2. Related Work

2.1 Existing Environments to Develop Numerical Codes

Various environments already exist for either generating efficient numerical codes or analyzing/proving the numerical properties of given programs. But, like any software, such tools have their own limitations. One can cite:

- *Spiral.*[1] This project develops a program generation system capable of producing C or Verilog code implementing digital signal processing algorithms in (complex or real) floating-point arithmetic, like DFTs [27, 28], and more recently, some non-linear transforms as well [15]. Given some architectural features (like, typically, various forms of parallelism: SMP, SIMD, etc.), the Spiral system automatically optimizes codes for efficiency. However, although overflows are avoided in some specific applications as described in [8], the generation process does not seem to implement error-analysis techniques.
- *Fluctuat.*[2] This software is a static analyzer devoted to the study of the propagation of rounding errors inherent to floating-point computations [10, 16, 29]. Given a (possibly very large) numerical code, Fluctuat can detect automatically catastrophic losses of accuracy and identify the corresponding faulty variable(s). Also, guaranteed error bounds are computed using affine arithmetic [2, 7]. However formal proofs are not produced and since Fluctuat is not freely available,[3] its extension is an issue.
- *Why.*[4] This platform for code certification is based on the idea that properties of some existing code can be proven by annotating the code through comments. Then these comments are turned into different proof obligations. Afterwards, one can deal with each proof obligation using an appropriate tool. Frama-C,[5] a C source static analyzer, has a plug-in named Jessie which converts C code with comments written in the ANSI/ISO C Specification Language[6] (ACSL) into Why code, thus allowing to reuse all the features of the Why platform.

  The main limitation of this approach lies in its restriction to one particular programming language. Frama-C for instance only works on preprocessed C code and will not yield a proof for some extended class of C implementations. Moreover, working on a specific code, rather than at the mathematical level, has two drawbacks. First, useful information could have been lost when the developer has implemented his algorithm. Second, it is conflicting with our idea to go from a mathematical description of a problem to code generation.

2.2 Tools Commonly Used for Specific Tasks

On the other hand, to assist the design of fast and provably-accurate implementations of mathematical functions, the following tools can be extremely useful. However, as we shall see, each of them is used to perform very specific tasks, so that the programmer usually has to combine them in order to get a full design flow:

- *Computer algebra systems such as Maple.*[7] Symbolic computation can be useful to simplify symbolic expressions or to obtain mathematical properties like the monotonicity of a given function.
- *Sollya.*[8] The implementation of mathematical functions is often performed by means of polynomial approximation. Sollya is able to compute such an approximation. In addition, one can ask for a guaranteed upper bound on the supremum norm of the difference between a function and its polynomial approximant, thus allowing to control the error committed at this approximation step.
- *Gappa,*[9] "a tool intended to help verifying and formally proving properties on numerical programs dealing with floating-point or fixed-point arithmetic."

[1] `http://spiral.net/`

[2] `http://www-list.cea.fr/labos/gb/LSL/fluctuat/index.html`

[3] Still, academics can ask Fluctuat's developers to obtain it.

[4] `http://why.lri.fr/` as well as [11], [12].

[5] `http://frama-c.com/`

[6] `http://frama-c.com/download/acsl_1.4.pdf`

[7] `http://www.maplesoft.com/Products/Maple/`

[8] `http://sollya.gforge.inria.fr/`

[9] `http://gappa.gforge.inria.fr/` as well as [22], [6], [4].

- *CGPE.*[10] This tool returns some efficient and certified evaluation schemes for univariate and bivariate polynomials, optimized for a specific target architecture. Even though code latency is the main focus, one can ask that the error due to the evaluation order does not exceed a given bound.
- *Tools to search for "worst cases".* [19] Such tools target specifically the correct rounding of mathematical functions in a fixed precision, mainly double precision. From the input data (precision, function, tested interval, etc.), these tools generate C code that performs the search. The most important part related to code generation is here a hierarchical approximation of a high-degree polynomial by degree-2 polynomials on consecutive sub-intervals, using a modulo-1 fixed-point arithmetic (easily emulated with the `mpn` layer of GMP), where each variable has its own precision (which must be a multiple of 32 or 64 bits, depending on the target). Even though dynamical error analysis is done to automatically determine the precision of each variable, all the proofs have been done manually. Some optimizations for current architectures or for specific functions would need to redo most of the error analysis and modify a large part of the tools. This problem was actually at the origin of LEMA, and its need to support some exotic arithmetics.
- *GNU MPFR.*[11] This library offers floating-point arithmetic in arbitrary precision with correct rounding. Here, arbitrary precision means that the precision is a parameter of each function call. Note that this library underlies the computations of the software tools mentioned in this section.

Most of these tools have been routinely used to produce and validate significant parts of the codes of mathematical software libraries like CRlibm [3] and FLIP [13].

2.3 Tool Integration Through LEMA

When implementing an application with certified numerical properties, one can in turn use some of the tools that have been previously presented. Note that they do not act on the same type of data: the information needed by a proof assistant may not be relevant to a general purpose computer algebra system, and a polynomial specialized tool may not be able to deal with abstract mathematical properties. As there certainly is a natural sequence of calling these tools from the mathematical level to the hardware one, we could have inserted small translators between each application in order to automate the whole process. But some steps would also need data of the specification not produced by its predecessor. For instance, the hardware instruction set and the characteristics of each useful instruction are needed when generating efficient code but not before. Indeed, it is unlikely that a computer algebra system would process them. Conversely, knowledge of the abstract level may be relevant at any time: the mathematical properties often help to choose the most efficient algorithm at the implementation stage. For this reason, we want to gather information produced by all the tools and to store them along with the specifications of the problem.

Figure 1 shows how we intend, in the long term, to address the issue of certified numerical code generation using LEMA: The user provides a description of a problem written in LEMA; then this description is enriched by interaction with the previously mentioned tools; finally, when enough information has been gathered, the user can proceed with the generation of code, possibly with proof.

Generated code can be in C language, possibly with function calls to some libraries, such as GNU MPFR if multiple precision is needed. Other targets are possible, such as the GCC middle-end, VHDL, or a proof assistant language.

Interactions with these tools (represented as arrows in Figure 1) use their native script languages.

3. Requirements Analysis

We analyze in this section the requirements of a language that must be sufficiently rich to describe numerical algorithms without loss of information.

We could have reused the annotation languages (like ACSL) that have been presented in the previous section but they are quite general and have only limited support for floating-point arithmetic. Instead of extending one of them, we propose a single language for all data. Such data are either problem specifications provided by the user, or properties generated by some tools (e.g., by a tool doing value range propagation or error analysis); they can also be some information that does not belong to the specifications. The latter case includes hints for the proof assistant provided by the user because of the difficulty of finding them in an automatic way. The language should hold the status of these data: how they have been added, whether they need to be checked by some prover, whether and how they have been checked, and so on.

A consequence of integrating all kind of information in one place is that this language must mix different domains: abstract mathematics, floating-point arithmetic, and hardware capacities. In each domain, we want to express as simply as possible some basic and high level properties.

In the mathematical domain, it shall be possible to express any kind of mathematical expression or relation that a computer algebra system can understand. The way expressions are written must preserve their intrinsic parallelism, that is, their independence with respect to an order of evaluation. This is another reason to exclude annotations in a procedural language.

In the floating-point domain, we want to express simple properties like exactness in a computation, error bound of an evaluation, or the range of a floating-point function, and there shall be possible to bind such properties to their proofs. To state initial specifications, we need a higher level of expressiveness to describe

- various arithmetics, including arbitrary precision, but also particular arithmetics such as fixed point modulo 1 in arbitrary precision (needed to generalize the hierarchical approximations of a function by polynomials, as described in [19]);
- the rounding mode and its properties;
- the measure of error as a function of the exact value and its approximation;
- how standards like IEEE 754-2008 [17] and C99 [18] define the behavior of functions on special values;
- formal proofs, possibly with properties declared as hypotheses (or axioms) if it is too difficult or impossible to prove them with current tools.

[10] `http://cgpe.gforge.inria.fr/` and [30].

[11] `http://www.mpfr.org/` and [14].

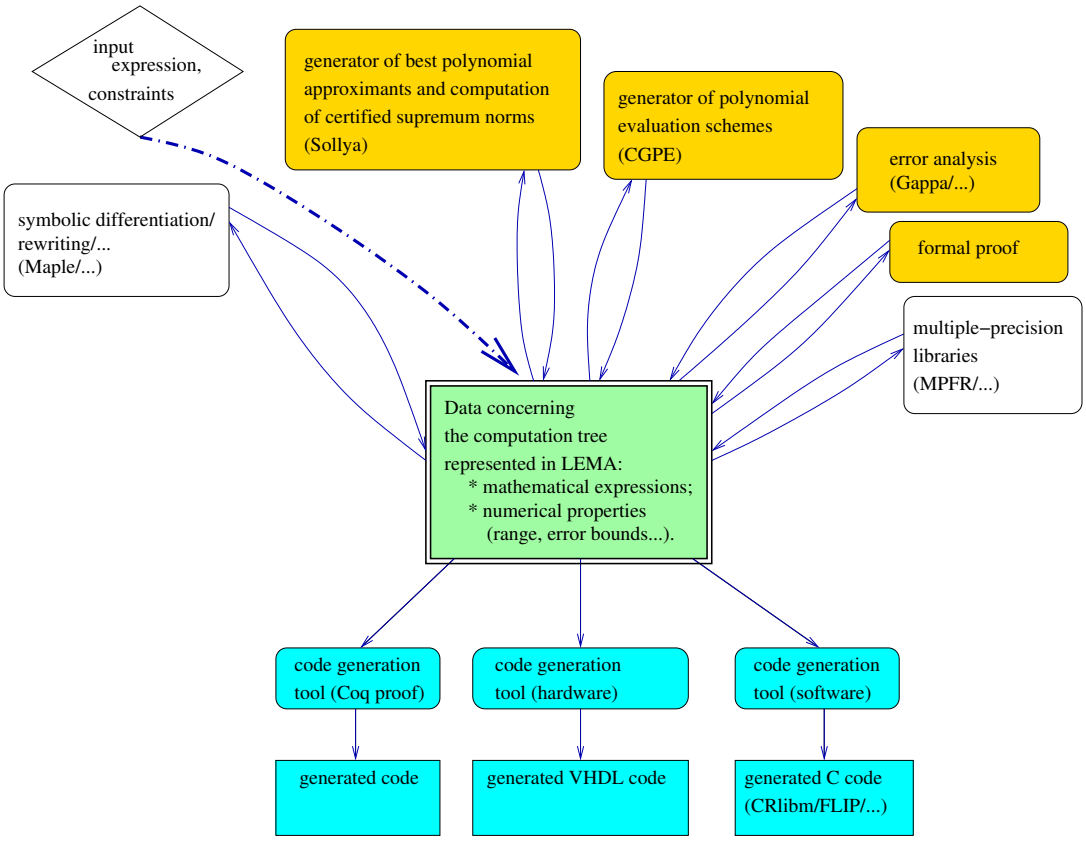

Figure 1. Use of LEMA for the process of certified numerical code generation.

In the hardware description domain, the useful information ranges from which integer and floating-point formats are available on the target platform, to a more complex level for the description of the instruction set characteristics in terms of execution time and parallelism capabilities.

And most of all, the language must provide innate extensibility, so that we can create it in a minimal form and enhance it progressively in response to our needs.

4. Rationale for XML and MathML

The first thing we need to express is numerical expressions. For the moment, let us ignore problems related to rounding. The problem that needs to be solved is generally expressed in a mathematical form, and we want to remain close to this form. This is better for the understanding of the computations and for the proof. For this reason, imperative style such as in C or FORTRAN with variables that can be reused in control structures such as loops was rejected. A functional style is highly preferable. We considered using XML as a basis for our language, since

- it allows us to define the features we want, including clean extensibility (e.g., via namespaces);
- it natively expresses trees, thus perfectly fits to the functional style;
- the use of decorations permits the representation of properties parametrized by the arithmetic, without duplication (see Figure 2 for an example);
- it already has various validating parsers, making this solution rather robust (validation could be seen as the first step of a proof: we ensure that what the user writes has an unambiguous meaning, and the various tools working on this language will always get a sane input).

Then the question is: how do we write numerical expressions, which are in fact mathematical expressions? We could define our own XML elements for that, but since specification work has already been done with the XML-based language MathML, we choose to reuse it and, in particular, the content markups. As a bonus, this allows to interoperate with other tools that understand MathML, though this is not the primary motivation.

Nevertheless, the content markup is not intended to express floating-point concepts and we need to extend it for this purpose. As we will see in the following section, we choose to extend MathML using the XML syntactical extensions. An alternative design would have been to define

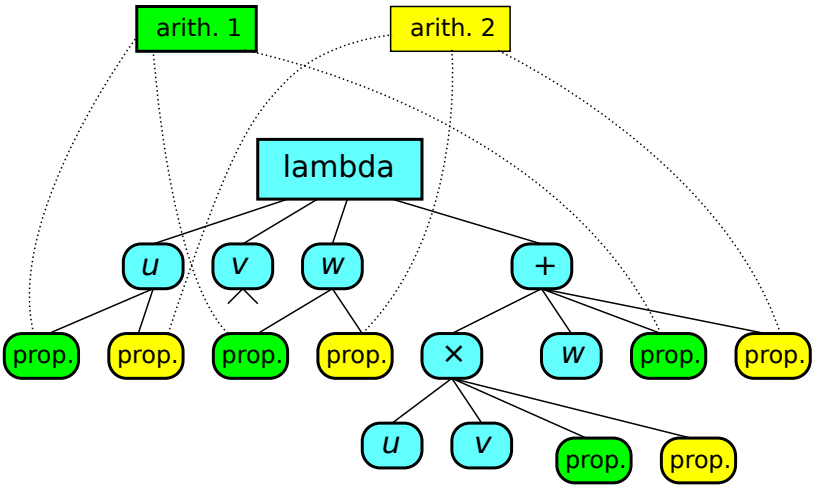

Figure 2. LEMA tree with properties depending on arithmetics.

new symbols in OpenMath content directories [26], but it leads to heavier and less manageable documents.

5. Annotating MathML Expressions With Evaluations

In this section, we present how we have extended the MathML language in order to support floating-point evaluations.

As a simple example, let us evaluate the interval

$$[0.17, 107148111696065103375347396388111517442326528]$$

in single precision (that is, in the binary32 format defined in the IEEE 754-2008 standard [17]) with rounding to nearest mode and even-rounding rule for the halfway cases. The exact mathematical interval we choose can be written in pure MathML as follows:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
 <interval>
  <cn id="left" type="real">0.17</cn>
  <cn id= "right" type="integer">
   107148111696065103375347396388111517442326528
  </cn>
 </interval>
</math>
```

Here, both endpoints are numbers for briefness, but they could be more sophisticated expressions as well, like polynomials or rational functions. Using a tool like Gappa, we can produce certified evaluations in any floating-point context and then include this information in the previous document. From a lexical point of view, what we want to provide in XML consists of

- means to bind evaluated values to the exact value from which they stem,
- means to record floating-point properties of these evaluations,
- means to attach certificates to evaluations.

From an efficiency point of view, we want each expression to be evaluated only once in a given floating-point context. Consequently, evaluated values have to be easy to find knowing the exact value node, and vice versa. Additionally, evaluated values must be recorded in such a way that they can be sent to external tools in text form with minimal manipulation.

5.1 Numbers in MathML

First, let us review the support of numbers defined in the content markup section of MathML 3.0 (see §4.2.1 "Numbers `<cn>`" in [20]). As we can see in the above example, the content number element `<cn>` encodes numbers and it specifies their kind with the `type` attribute. In strict MathML, the `type` attribute may only take four values: `integer`, `real`, `double`, and `hexdouble`, the last two ones being dedicated to floating-point numbers in double precision. This restriction to a unique precision prevents us to use them as a general means for floating-point number encoding. Numbers of the `real` type are written as "an optional sign (+ or −) followed by a string of digits possibly separated into an integer and a fractional part by a decimal point. Some examples are 0.3, 1, and −31.56." The drawback with this type is the absence of exponent, which hinders its use for numbers whose magnitude is either tiny or huge. In non-strict MathML, floating-point numbers can be encoded with the `e-notation` type. For instance, `<cn type="e-notation">12.3<sep/>3</cn>` represents 12.3 times 10^3. Note also that in non-strict MathML a `base` attribute can be used to specify the base in which the text content of the `cn` element should be interpreted. This `e-notation` type allows writing floating-point numbers in arbitrary precision but it requires a conversion before the number can be read by non MathML-aware tools.

Furthermore, the MathML vocabulary can be enriched using symbols defined in OpenMath content dictionaries and, indeed, the content dictionary bigfloat1.cd of [5] defines a general floating-point representation of numbers. For example, the following sample encodes the floating-point value $12.625 = 101 \cdot 2^{-3}$:

```
<apply>
 <csymbol cd="bigfloat1">bigfloat</csymbol>
  <cn type="integer">101</cn>
  <cn type="integer">2</cn>
  <cn type="integer">-3</cn>
</apply>
```

Again, a number encoded this way has to be converted before being sent in text form to external tools.

5.2 Floating-Point Numbers in LEMA

To address the problem of consecutive conversions, we choose a floating-point number representation stricter than the one specified in the IEEE-754 standard (see §5.12.3 "External hexadecimal-significand character sequences representing finite numbers" in [17]): first, the mandatory sign, followed by the `'0x'` prefix, and the hexadecimal integral significand with digits in lower case; second, the binary exponent in the following form: the exponent indicator `'p'`, the mandatory exponent sign, and at last the exponent value written in decimal. For instance, the right endpoint value in our example

```
1071481116960651033753473963881151744232652 8
```

is written in this format as

```
+0x7bp+136.
```

This allows us to represent binary floating-point numbers of any precision. It will be compatible with any tool that reads floating-point inputs with the C standard library, or multiple precision inputs with the GNU MPFR library.

Since such an encoding is not available in MathML, we define in a new namespace the special attribute `lema:type` of the `cn` element. The values of `lema:type` are custom floating-point types like "Binary32" for a floating-point number belonging to the set defined by the binary32 format of the IEEE 754-2008 standard [17] and they indicate that the text content of `cn` should be interpreted as a number written in the form described above.

The same LEMA namespace is also used for any other floating-point properties not already defined in MathML. We define some additional attributes for the `cn` element: the floating-point format and the rounding mode are specified in separate attributes called `lema:type` and `lema:rounding`, respectively. This helps filter the document for a particular precision or rounding mode. Moreover, each of the values `lema:type` and `lema:rounding` is used as a suffix to the value of the initial `id` attribute in the evaluated number `id`, so that the connection between the initial value and its evaluation is immediate to a human reader. This entails some data duplication but consistency is easy to check. Finally, the boolean attributes `lema:exact` and `lema:overflow` indicate, respectively, if the evaluation is exact[12] and if it overflows. The example may be rewritten as follows:

```
<cn id="right_Binary32_Nearest"
 lema:type="Binary32"
 lema:rounding="Nearest"
 lema:exact="true"
 lema:overflow="true">+0x7bp+136</cn>
```

5.3 Annotating with Floating-Point Evaluations

Whereas we had to extend the MathML number encoding, we can reuse the mechanism provided by MathML to annotate elements with application specific information: the pair `<semantics>`, `<annotation-xml>` of elements (see §4.2.8 "Attribution via `semantics`" in [20]).

The `semantics` element is a container whose first child is the expression being annotated and whose other children are the annotations, each annotation being enclosed in an `annotation` or `annotation-xml` element. By this means, it is possible to provide several alternative presentations of the expression or to change its mathematical meaning with additional information. We use this annotation system to attach an evaluated value to the exact value from which it is derived. This evaluated value can be seen as an interpretation of the mathematical value in a given floating-point context. The `encoding` attribute of an `annotation` element indicates the data format of its text contents; for our needs we use the `application/lema-evaluation+xml` value, thus following the OpenMath example which uses `application/openmath+xml` and the recommendations of RFC 3023 for XML Media Types [25]. Therefore, our example with the evaluations can be written as follows:

```
<math xmlns="http://www.w3.org/1998/Math/MathML"
 xmlns:lema="http://www.ens-lyon.fr/LIP/Arenaire/lema">
 <interval>
  <semantics>
   <cn id="left" type="real">0.17</cn>
   <annotation-xml lema:type="Binary32_Nearest"
    encoding="application/lema-evaluation+xml">
    <cn id="left_Binary32_Nearest"
     lema:type="Binary32"
     lema:rounding="Nearest"
     lema:exact="false">+0xae147bp-26</cn>
   </annotation-xml>
  </semantics>
  <semantics>
   <cn id= "right" type="integer">
    1071481116960651033753473963881151744232652 8
   </cn>
   <annotation-xml lema:type="Binary32_Nearest"
    encoding="application/lema-evaluation+xml">
    <cn id="right_Binary32_Nearest"
     lema:type="Binary32"
     lema:rounding="Nearest"
     lema:exact="true"
     lema:overflow="true">+0x7bp+136</cn>
   </annotation-xml>
  </semantics>
 </interval>
</math>
```

As a `semantics` element admits several `annotation-xml` children, we can attach to a single number many evaluations in different floating-point contexts. Contexts are differentiated with the `lema:type` attribute of the `annotation-xml` element, which eases the retrieval of a given evaluation by just browsing among siblings of the exact number node. Thanks to this proximity, the converse operation, that is, finding the exact value knowing the evaluated value node, is simple too.

5.4 Annotating Evaluations with Certificates

Furthermore, we would like to certify, typically by using Gappa, evaluated values and properties such as exactness. In the following example, we present an excerpt where such a Gappa proof is embedded after the evaluated value using the ability of the `annotation-xml` element to contain application-specific elements:

```
<semantics>
 <cn id= "right" type="integer">
  1071481116960651033753473963881151744232652 8
 </cn>
 <annotation-xml lema:type="Binary32_Nearest"
  encoding="application/lema-evaluation+xml">
```

[12] The notion of exactness is defined regardless of the range of the exponent.

```
  <cn id="right_Binary32_Nearest"
   lema:type="Binary32"
   lema:rounding="Nearest"
   lema:exact="true"
   lema:overflow="true">+0x7bp+136</cn>
  <lema:proof href="right_Binary32_Nearest"
   type="gappa">
<![CDATA[
@rndn = float< 24, -126, ne >;
MaxFloat = 0xf.fffffp+124;
right = 1071481116960651033753473963881151744232652 8;
right_Binary32_Nearest = +0x7bp+136;

{
 right_Binary32_Nearest - rndn(right) in [0, 0]
 /\ right_Binary32_Nearest - right in [0, 0]
 /\ right_Binary32_Nearest - MaxFloat >= 0
}
]]>
  </lema:proof>
 </annotation-xml>
</semantics>
```

Here we have introduced a new `lema:proof` element as a container for the script proving the evaluation whose identifier is referenced by the `href` attribute. Several proofs for the same evaluation but written in different tool languages can be embedded at this point, and the `type` attribute differentiates them. Here, the CDATA content is a script that Gappa can interpret. A Gappa script is composed of two or three sections: The first one is where symbols are defined; the second one, written between curly brackets, is a logical formula to be proven; the last one, which is optional, is a series of hints like rewriting rules or bisection directives, and is aimed at helping the Gappa engine (see [21] for further information).

In the above example, the first four lines in the CDATA part belong to the definition section: the symbol `rndn` defines both the rounding mode and the floating-point format, while `MaxFloat` corresponds to the maximal number that can be represented in the binary32 format. The values of the `id` attributes are reused to define symbols for the corresponding numbers, making it easier to map them to their counterpart in the XML document.

The lines between curly brackets form the logical formula, which is a conjunction of statements. The first line states that the rounded value truly is +0x7bp+136, the second one, that the exact value of the right endpoint is represented without rounding error, and the last one that it actually overflows.

From this script, Gappa can also derive formal proofs in Coq and HOL Light, which could be embedded next to it. But as the formal proofs are mere certificates that are not used thereafter and that should not be corrupted by subsequent transformations, it is preferred to save them in external files, using the `src` attribute value to record their URI, as in:

```
<semantics>
 <cn id="left">0.17</cn>

 <annotation-xml lema:type="Binary32_Nearest"
  encoding="application/lema-evaluation+xml">
  <cn id="left_Binary32_Nearest"
   lema:type="Binary32"
   lema:rounding="Nearest"
   lema:exact="false">
   +0xae147bp-26
  </cn>
  <lema:proof href="left_Binary32_Nearest"
   type="gappa"
   src="left_Binary32_Nearest.gappa"/>
  <lema:proof href="left_Binary32_Nearest"
   type="coq"
   src="left_Binary32_Nearest.v"/>
 </annotation-xml>
</semantics>
```

This shortens somewhat the XML document, as Coq proofs are *very* lengthy. In addition, if all proofs are saved in a single directory, it is easy to check them all by sending them in a row to the proof checker.

6. Conclusion and Perspectives

In this paper, we have presented the LEMA language, our extension of MathML that suits our needs for code generation and formal proofs of numerical codes. We have exemplified the extension to floating-point arithmetic: representation of floating-point numbers and creation of the `lema:type` attribute, incorporation of their evaluation in floating-point arithmetic through annotations, expressions of floating-point properties through attributes and links to their proof.

Beyond floating-point evaluation, LEMA should express other arithmetic properties like error bounds and value ranges, special values (infinity, signed zero, NaN), and arithmetics and their associated formats. This is currently under development. At a higher level, specifications, such as a description of hardware capabilities or a function specification, can be reused. Thus, we intend to set up a database mechanism. Meanwhile, we are elaborating an XML schema to validate LEMA documents.

The LEMA language is developed simultaneously with a library that enables to integrate various tools, as shown in Figure 1. Linking with the tools other than Gappa is work in progress.

Acknowledgments

This work is supported by the ANR project EVA-Flo.

References

[1] R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Mar. 2010. URL `http://www.loria.fr/~zimmerma/mca/mca-cup-0.5.1.pdf`. Version 0.5.1.

[2] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *VI Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens (SIBGRAPI'93)*, pages 9–18, Oct. 1993.

[3] CRlibm. CRlibm (Correctly Rounded mathematical library). URL `http://lipforge.ens-lyon.fr/www/crlibm/`.

[4] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1), 2009.

[5] J. Davenport. OpenMath Content Dictionary: bigfloat1, 1999. URL `http://www.openmath.org/cd/bigfloat1.ocd`.

[6] F. de Dinechin, C. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, Dijon, France, 2006. URL `http://www.msr-inria.inria.fr/~gmelquio/doc/06-mcms-article.pdf`.

[7] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.

[8] F. de Mesmay, S. Chellappa, F. Franchetti, and M. Püschel. Computer generation of efficient software Viterbi decoders. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, volume 5952 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2010.

[9] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[10] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[11] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003. URL `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[12] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV*, pages 173–177, Berlin, Germany, 2007.

[13] FLIP. FLIP (Floating-point Library for Integer Processors). URL `http://flip.gforge.inria.fr/`.

[14] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007.

[15] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009.

[16] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006.

[17] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008. ISBN 978-0-7381-5752-8. Available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[18] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:1999, Geneva, Switzerland, Dec. 1999.

[19] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Jan. 2000. URL `http://www.vinc17.net/research/papers/these.ps.gz`.

[20] MathML. Mathematical markup language (MathML) version 3.0. URL `http://www.w3.org/TR/2009/CR-MathML3-20091215/`.

[21] G. Melquiond. *User's Guide for Gappa*. URL `http://gappa.gforge.inria.fr/doc/index.html`.

[22] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2006. URL `http://www.msr-inria.inria.fr/~gmelquio/doc/06-these.pdf`.

[23] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[24] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[25] Network Working Group. XML Media Types, 2001. RFC 3023.

[26] OpenMath2. The openmath standard, version 2.0, 2004. URL `http://www.openmath.org/standard/om20-2004-06-30/omstd20html-0.xml`.

[27] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[28] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[29] S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Novel Approaches to Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 295–312, 2004.

[30] G. Revy. *Implementation of binary floating-point arithmetic on embedded integer processors: polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, Dec. 2009.