Provably Faithful Evaluation of Polynomials

Sylvie Boldo, César Muñoz

Provably Faithful Evaluation of Polynomials*

Sylvie Boldo
INRIA
LRI, Bâtiment 490
Université Paris-Sud
91405 Orsay Cedex, France
sylvie.boldo@inria.fr

César Muñoz
National Institute of Aerospace
100 Exploration Way
Hampton, Virginia, 23666, USA
munoz@nianet.org

ABSTRACT

We provide sufficient conditions that formally guarantee that the floating-point computation of a polynomial evaluation is faithful. To this end, we develop a formalization of floating-point numbers and rounding modes in the Program Verification System (PVS). Our work is based on a well-known formalization of floating-point arithmetic in the proof assistant Coq, where polynomial evaluation has been already studied. However, thanks to the powerful proof automation provided by PVS, the sufficient conditions proposed in our work are more general than the original ones.

Categories and Subject Descriptors

B.2.3 [**Hardware**]: Arithmetic and Logic Structures—*Reliability, Testing, and Fault-Tolerance*; F.4 [**Theory of Computation**]: Mathematical Logic and Formal Languages

Keywords

Floating-point, polynomial evaluation, formal verification

1. INTRODUCTION

Many engineering applications rely on the accuracy of the numerical computations they perform, usually on floating-point numbers. As the Pentium Bug [6] illustrates, the correctness of floating-point arithmetic is a safety critical issue.

Traditionally, digital systems are validated via extensive testing and simulation. In the past years, an alternative set of techniques, known as *formal methods*, have been successfully applied to the specification and verification of a variety of systems, including both hardware-level and high-level floating-point arithmetic [1, 8, 12–15, 17, 19, 21, 22, 24, 26–28]. Formal methods are based on discrete mathematics and logic, and, in contrast to testing and simulation, they enable an exhaustive exploration of the set of possible states of a system, even when this set is infinite. Hence, formal methods can provide a *guarantee* that a system satisfy some safety requirements.

In this paper, we *formally* study the accuracy of polynomial evaluations that use floating-point arithmetic. This is particularly relevant to safety analysis of engineering applications as many numerical computations performed in these applications are polynomial evaluations. Furthermore, polynomial evaluation is an important part of the computation of elementary functions (such as exponential and trigonometric functions).

The main objective of this work is to provide sufficient conditions that *guarantee* that the floating-point computation of a polynomial evaluation is *faithful*. To this end, we develop a fairly complete formalization of floating-point numbers in the verification system PVS. Our work is based on a generic floating-point arithmetic library[1] designed by Daumas, Rideau, and Théry [8], and implemented in Coq. The application of a floating-point formalization to polynomial evaluation was already studied in [4] using the Coq-based floating-point library. However, the results obtained in our PVS formalization are significantly better than the original ones. This is mostly due to the proof automation provided by PVS.

The rest of this paper is organized as follows. Section 2 presents the formalization of floating-point numbers in PVS. Standard rounding modes and our notion of faithful rounding are defined in Section 3. In Section 4, we apply this formalization to polynomial evaluation. Finally, in Section 5 we present examples of faithful evaluations of polynomials. The mathematical development presented in this paper has been formally verified in PVS and it has been integrated to the NASA Langley PVS Libraries.[2]

2. FLOATING-POINT NUMBERS IN PVS

Floating-point arithmetic used in most general-purpose processors is described by the IEEE-754 [30, 31] and IEEE-854 [5] standards. These standards define the format, rounding modes, and operations that can be performed on floating-point numbers. For more information on floating-point num-

*This work was done while the first author was visiting the National Institute of Aerospace, supported by the National Aeronautics and Space Administration, Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043, and by the French National Center for Scientific Research, CNRS PICS No. 2533.

SAC'06 April 23-27, 2006, Dijon, France

[1]Available at `http://lipforge.ens-lyon.fr/projects/pff`.
[2]Available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

bers and numerical computation see [11, 16, 32].

We develop a PVS library of floating-point numbers based on a well-known generic floating-point library written in Coq [8]. That library is especially useful when dealing with high-level algorithms (see, for example, [2]) because it does not consider the machine-level array of bits, but only integer numbers that are more easily handled by a person or a proof assistant. The same approach was used beforehand by Harrison [12–15].

Coq [7] and PVS [25] are comparable proof assistants based on higher-order logic. They both offer very expressive specification languages and highly sophisticated theorem provers. However, PVS has a well-deserved reputation of providing powerful proof automation tools in the form of decision procedures. In this paper, we exploit these features of PVS to improve and extend the original development in Coq.

2.1 Floats

Following the definition in [8], a floating-point number is represented by a pair of integers, e.g., the radix-2 floating-point number $1.001_2\text{E}1$ is represented as $(1001_2, -2)$, i.e. $(9, -2)$. The left part of a float is called the *significand* and the right part is the *exponent*. Note that the exponent is shifted compared to the exponent of the IEEE machine number. In PVS, we declare the type `float` as

```
float : TYPE = [# Fnum:int, Fexp:int #]
f,g,h : VAR float
r     : VAR real
```

In this paper, we use PVS boxed declarations and statements, rather than mathematical notation, to emphasize the fact that this specification was written in a formal notation and that all lemmas and propositions have been mechanically checked in a proof assistant. PVS notation is similar to the notation used in functional programming languages and for most part is self explanatory. In this case, we say that `float` is a record type with fields `Fnum` and `Fexp` that correspond to the significand and the exponent, respectively. For instance, a floating-point number f with significand n and exponent e is written in PVS as `(# Fnum :=` n`, Fexp :=` e `#)`. The name of the fields are also the selectors, e.g., `Fnum(`f`)` is the significand n and `Fexp(`f`)` is the exponent e. Furthermore, as PVS is a fully-typed language, we declare the mathematical variables `f`, `g`, and `h` of type `float`, and `r` of type `real`. These variables are later used as parameters in function definitions and as implicitly quantified universal variables in lemmas and theorems.

The radix is defined as 2 in the IEEE-754 standard and can be either 2 or 10 in the IEEE-854 standard. The radix β (`radix`, in PVS) is here a parameter of the specification and it is declared as an integer greater than 1. Therefore, a float can be interpreted as a real value as follows:

$$(n, e) \in \mathbb{Z}^2 \quad \hookrightarrow \quad n \times \beta^e \in \mathbb{R}$$

```
FtoR(f):real = Fnum(f)*radix^(Fexp(f))
CONVERSION FtoR
```

We declare FtoR as a *conversion*. This way, elements of type `float` are automatically converted into real numbers when needed. For example, if f and g are of type `float`, then f`+`g is automatically converted to `FtoR(`f`)+FtoR(`g`)` because the addition operation `+` is, by default, defined over real numbers.

2.2 Bounded Floats

The type `float` represents an infinite number of numbers and only a finite of these can be represented as machine floating-point numbers. We have to restrict this type to the numbers that fit in a given floating-point format. A floating-point format (typically IEEE single or IEEE double precision) is a pair of integers (p, E). The integer p is called the *precision* of the floating-point format and E is the *minimal exponent*. For example, the IEEE double precision is specified by the pair $(53, 1074)$ and the single precision is specified by the pair $(24, 149)$. For a given format (p, E), we say that a float (n, e) is *bounded* if and only if

$$|n| < \beta^p \quad \text{and} \quad -E \leq e.$$

```
Format: TYPE = [# Prec:above(1), dExp:nat #]
b     : VAR  Format
vNum(b):posnat = radix^Prec(b)

Fbounded?(b)(f):bool =
  abs(Fnum(f))<vNum(b) AND -dExp(b) <= Fexp(f)
```

The lower bound on the exponent is needed as it creates subnormal numbers, whose behavior is often unexpected. In our formalization, we do not consider overflows and we argue that they can be handled at a higher specification level. Overflows create infinities and NaNs, but they are usually propagated until the end of a computation. Therefore, overflows are more easily detected than underflows as subnormal numbers are silent even when the loss of accuracy is significant.

2.3 Canonical Floats

The representation of floats in this formalization is redundant, i.e., several floats may have the same real value. This is true even if the floats are bounded. For example, using radix 2 and 4 bits of precision, the floats $(8, 0)$, $(4, 1)$, $(2, 2)$, and $(1, 3)$ are all bounded and have the real value 8. The sets of floats that share the same real value are called a *cohort*.

In order to represent IEEE machine floating-point numbers, which are unique, we have to define a canonical set of floats. A *canonical float* is a float that is either normal or subnormal. A *normal float* is a float such that its significand cannot be multiplied by the radix and still fit in the format. This means that the first digit of the significand, represented in base β, is non-zero. A *subnormal float* is a float having the minimal exponent such that its significand could be multiplied by the radix and still fit in the format.

By definition, normal and subnormal floats are disjoint. Subnormal floats are the smallest representable floats (in absolute value) and their characteristics are very different from the normal floats. They may produce surprising numerical results due to their uncommon characteristics.

```
Fnormal?(b)(f):bool =
  Fbounded?(b)(f) AND vNum(b)<=abs(radix*Fnum(f))

Fsubnormal?(b)(f):bool =
  Fbounded?(b)(f) AND Fexp(f)=-dExp(b) AND
  abs(radix*Fnum(f))<vNum(b)

Fcanonic?(b)(f):bool =
  Fnormal?(b)(f) OR Fsubnormal?(b)(f)
```

We prove that canonical floats are unique: if two floats are canonical and have the same real value, then they are identical. We show that any bounded float has a canonical representation obtained by applying the function `Fnormalize`.

```
FcanonicUnique: Lemma
  Fcanonic?(b)(f) AND Fcanonic?(b)(g) AND
  FtoR(f)=FtoR(g)
    => f=g

Fnormalize(b)(f:(Fbounded?(b))): recursive
  {x : (Fcanonic?(b)) |
     FtoR(x) = FtoR(f) AND Fexp(x) <=  Fexp(f)} =
       ...
```

The definition of the function `Fnormalize` have been shortened for the sake of simplicity. The main difference with the Coq definition is that we take advantage of several PVS features. First, it declares the parameter `f` as a bounded float, whose definition depends on the format `b`. This feature is called *dependent types*. Second, the range of the function is declared as a canonical float `x` that satisfies: `FtoR(x)=FtoR(f) AND Fexp(x)<=Fexp(f)`. This feature is called *predicate sub-typing*. The typing mechanism of PVS makes sure that every time the function `Fnormalize` is used the parameter `f` is of type `Fbounded?(b)`. Moreover, the PVS type system guarantees that the result of the function `Fnormalize` satisfies the given predicate sub-typing.

We also prove that the canonical representation is the one having the smallest exponent of the cohort. The concept of many floats (a cohort) for one real value has been addressed in the revision of the IEEE-754 standard where it is especially interesting for radix-10 numbers.

```
CanonicLeastExp: Lemma
  Fcanonic?(b)(f) AND Fbounded?(b)(g) AND
  FtoR(f)=FtoR(g)
    => Fexp(f) <= Fexp(g)
```

Furthermore, we show that given two non-negative IEEE floating-point numbers f and g, f is smaller than g if the string of bits representing f is less, in lexicographical order, than the string of bits representing g. In our formalization, we express that as the fact that the real value and the exponent of two positive floats are in the same order relation.

```
Lexico: Lemma
  Fcanonic?(b)(f) AND Fcanonic?(b)(f) AND
  0 <= f AND f <= g
    => Fexp(f) <= Fexp(g)
```

2.4 Predecessor, Successor and Ulp

The predecessor of a given float f is the greatest float strictly less than f and is denoted by f^-. The successor of a given float f is the smallest float strictly greater than f and is denoted by f^+. The significand and exponent of both the predecessor or the successor of a float f can easily be deduced from f. The definition of the functions `Fsucc` and `Fpred` are found in the PVS files.

We prove several useful properties of these functions. For example, the opposite of the successor is the predecessor of the opposite (`FpredFoppFsucc`) and the fact that a positive float has a non-negative predecessor (`FpredPos`):

```
FpredFoppFsucc: Lemma
  Fpred(b)(Fopp(f)) = Fopp(Fsucc(b)(f))

FpredPos: Lemma
  Fcanonic?(b)(f) AND 0 < f
    => 0 <= Fpred(b)(f)
```

The *unit in the last place (ulp)* is the value of the least significant digit of the IEEE representation of the float. It is also the increment to add to a positive float to get the successor of this float. The ulp can be defined as the radix to the power of the exponent, if the float is canonical.

```
Fulp(b)(f:(Fbounded?(b))):real =
  radix^(Fexp(Fnormalize(b)(f)))
```

We use here Goldberg's definition of the ulp [11]. There have been several different definitions of ulp depending on the author. See [23] for more details on the possible definitions and properties, especially for generic radices.

3. ROUNDING MODES

Floating-point operations in the IEEE standards are defined such that the result is the same as if the operation is computed with an infinite precision and then rounded to the destination format. Hence, instead of a direct definition of *floating-point addition* $\oplus$, we define a rounding mode $\circ$ over real expressions, and from there we could define the floating-point addition based on $\circ(f + g)$. The IEEE standards require several possible definitions of the rounding operation $\circ$. For instance, the *rounding toward* $-\infty$ (`isMin?`, in PVS) is the biggest floating-point number whose value is smaller than the real number. Similarly, the *rounding toward* $+\infty$ (`isMax?`, in PVS) is the smallest bounded floating-point number whose value is bigger than the real number.

As several bounded floats may be correct roundings of the same real number, we define the type of rounding operations, `RND` in PVS, as a relation between real numbers and bounded floats, indexed by a format, rather than a function from real numbers to bounded floats.

```
RND : TYPE =
  [b:Format -> [[real,(Fbounded?(b))]->bool]]
P   : VAR RND

isMin?(b)(r:real,min:(Fbounded?(b))):bool =
  min <= r AND
  Forall (f:(Fbounded?(b))): f <= r => f <= min

isMax?(b)(r:real,max:(Fbounded?(b))):bool =
  r <= max AND
  Forall (f:(Fbounded?(b))): r <= f => max <= f
```

We do not explain (yet) how to compute these rounding modes, we just state the properties they satisfy. Furthermore, we say that a rounding mode is *well-defined* if it is

- *total*, i.e., all reals can be rounded,
- *compatible*, i.e., if two floats have the same real value and one is a rounding of a real value, then the other one is too,
- *minormax*, i.e., each rounding is either the rounding toward $+\infty$ or $-\infty$ of the real number, and

- *monotone*, i.e., non-decreasing.

Some rounding modes are moreover *unique*, i.e., each real number has only one rounding, which may have a cohort of representations.

```
Total?(b)(P):bool = Forall (r:real):
  Exists (f:(Fbounded?(b))): P(b)(r,f)

Compatible?(b)(P):bool =
  Forall (r1,r2:real, f1,f2:(Fbounded?(b))):
    P(b)(r1,f1) AND r1=r2 AND FtoR(f1)=FtoR(f2)
      => P(b)(r2,f2)

MinOrMax?(b)(P):bool =
  Forall (r:real,f:(Fbounded?(b))):
    P(b)(r,f) => isMin?(b)(r,f) OR isMax?(b)(r,f)

Monotone?(b)(P):bool =
  Forall (r1,r2:real, f1,f2:(Fbounded?(b))):
    r1 < r2 AND P(b)(r1,f1) AND P(b)(r2,f2)
      => f1 <= f2

Unique?(b)(P):bool =
  Forall (r:real,f1,f2:(Fbounded?(b))):
    P(b)(r,f1) AND P(b)(r,f2)
      => FtoR(f1)=FtoR(f2)

RoundedMode?(b)(P):bool =
  Total?(b)(P) AND Compatible?(b)(P) AND
  MinOrMax?(b)(P) AND Monotone?(b)(P)
```

Note that, in PVS, the fact that a rounding mode P is well-defined for a given format b, is written `RoundedMode?(`b`)(`P`)`.

3.1 IEEE Rounding Modes

The IEEE standards define 4 rounding modes, but a fifth one has been added in the revision of the IEEE-754 standard. We have already defined the rounding toward $\pm\infty$, we now add all the other rounding modes defined by the IEEE-754 standard and its revision. The *nearest* rounding mode yields the float that is nearer to the real value. Note that this rounding is not unique when the real number is exactly in the middle of two floats. The revision of the IEEE-754 standard defines two unique rounding modes to the nearest: a *nearest even* rounding mode (which when in the middle, chooses the float having an even significand), and a *nearest away from zero* rounding mode (which when in the middle, chooses the one with the greatest absolute value).

```
ToZero?(b)(r:real,f:(Fbounded?(b))):bool =
   if 0 <= r then isMin?(b)(r,f)
             else isMax?(b)(r,f) endif

Nearest?(b)(r:real,f:(Fbounded?(b))):bool =
   (Forall (g:(Fbounded?(b))):
      abs(f-r) <= abs(g-r))

EvenNearest?(b)(r:real,f:(Fbounded?(b))):bool =
   Nearest?(b)(r,f) AND
     (even?(Fnum(Fnormalize(b)(f))) OR
       (Forall (g:(Fbounded?(b))):
           Nearest?(b)(r,g) => FtoR(f)=FtoR(g)))
```

```
AFZNearest?(b)(r:real,f:(Fbounded?(b))):bool =
   Nearest?(b)(r,f) AND
     (abs(r) <= abs(f) OR
       (Forall (g:(Fbounded?(b))):
           Nearest?(b)(r,g) => FtoR(f)=FtoR(g)))
```

We prove that these rounding modes are well-defined and, except for the nearest rounding mode, that they are unique.

Finally, we provide functional specifications of the toward $\pm\infty$ and even nearest rounding modes, and prove that they are correct. The definition of these functions have been omitted here for the sake of simplicity.

Note that the IEEE standards only require correct rounding for $+$, $-$, $\times$, $/$, $\sqrt{}$, and for the fused multiply-and-add (FMA): $a \times b + c$ with only one rounding in the revision of the IEEE-754 standard. Therefore, although these rounding modes can be used to round any real number, e.g., exp(2), there is no guarantee that the result is the same as the floating-point computation of exp(2) on a particular processor.

3.2 Properties of Rounding Modes

A useful property of the rounding modes concerns the rounding of opposite numbers: for example, the rounding down of r is the opposite of the rounding up of $-r$.

```
MinOppMax: Lemma
  Fbounded?(b)(f) AND isMin?(b)(r,f)
    => isMax?(b)(-r,Fopp(f))

MaxOppMin: Lemma
  Fbounded?(b)(f) AND isMax?(b)(r,f)
    => isMin?(b)(-r,Fopp(f))

NearestFopp: Lemma
  Fbounded?(b)(f) AND Nearest?(b)(r,f)
    => Nearest?(b)(-r,Fopp(f))

NearestFabs: Lemma
  Fbounded?(b)(f) AND Nearest?(b)(r,f)
    => Nearest?(b)(abs(r),Fabs(f))
```

Another useful property is the fact that the sign of a real number is preserved by any rounding mode: a non-negative real is always rounded into a non-negative float.

```
RleRoundedR0: Lemma
  Fbounded?(b)(f) AND RoundedMode?(b)(P) AND
  P(b)(r,f) AND 0 <= r
    => 0 <= f

RleRoundedLessR0: Lemma
  Fbounded?(b)(f) AND RoundedMode?(b)(P) AND
  P(b)(r,f) AND r <= 0
    => f <= 0
```

Moreover, a bounded float is always rounded to itself.

```
RoundedProjectorEq: Lemma
  Fbounded?(b)(f) AND Fbounded?(b)(g) AND
  RoundedMode?(b)(P) AND P(b)(f,g)
    => FtoR(f)=FtoR(g)

RoundedProjector: Lemma
  Fbounded?(b)(f) AND RoundedMode?(b)(P)
    => P(b)(f,f)
```

3.2.1 Round-off Errors

The round-off error is the difference between the real value and its rounding. It is usually described in terms of the ulp (Section 2.4). For any rounding mode, this difference is strictly less than one ulp. And for any rounding to the nearest, this difference is less than or equal to half an ulp.

```
RoundedModeUlp: Lemma
  Fbounded?(b)(f) AND RoundedMode?(b)(P) AND
  P(b)(r,f)
    => abs(f-r) < Fulp(b)(f)

NearestUlp: Lemma
  Fbounded?(b)(f) AND Nearest?(b)(r,f)
    => abs(f-r) <= Fulp(b)(f)/2
```

3.2.2 Exact Subtraction

This property has been known for decades [29] but may be due to W. Kahan [18]. This theorem gives sufficient conditions for a subtraction to be exact. The theorem states that if f and g are bounded floats such that $\frac{f}{2} \leq g \leq 2f$, then a known float, which has the value $g - f$, is bounded.

```
Sterbenz: Theorem
  Fbounded?(b)(f) AND Fbounded?(b)(g) AND
  f/2 <= g AND g <= 2*f
    => Fbounded?(b)(Fminus(g,f))
```

By Lemma `RoundedProjector` (Section 3.2), a bounded float is exactly rounded. Therefore, the floating-point computation $\circ(g - f)$ is correct for any rounding mode $\circ$. Note that we have here exhibited a bounded float equal to $q - p$, which is not necessarily canonical (we can always normalize it afterward if needed). The way this lemma is stated makes it easy to use: instead of "there exists a bounded float such that ...", it gives a particular float that is bounded.

3.2.3 Representable Errors

It has been known since the 70s that the error of a floating-point addition (when rounding to the nearest) or multiplication fits in a floating-point number of the same format [9,20]. We give necessary and sufficient conditions for this error to be representable, even when underflow occurs [3]. Furthermore, we compute the exponent of the exhibited bounded float that represents the error term.

```
errorBoundedPlus: Lemma
  Fbounded?(b)(g) AND Fbounded?(b)(h) AND
  Fbounded?(b)(f) AND Nearest?(b)(g+h,f)
    => (Exists (e:(Fbounded?(b))):
        e=g+h-f AND Fexp(e)=min(Fexp(g),Fexp(h)))

errorBoundedMult: Lemma
  Fbounded?(b)(g) AND Fbounded?(b)(h) AND
  Fbounded?(b)(f) AND RoundedMode?(b)(P) AND
  P(b)(g*h,f) AND -dExp(b) <= Fexp(g)+Fexp(h)
    => (Exists (e:(Fbounded?(b))):
        e=g*h-f AND Fexp(e)=Fexp(g)+Fexp(h))
```

3.3 Faithful Rounding

A *faithful rounding* is a relation between a real number and a floating-point number such that the floating-point number is either the rounding up or the rounding down of the real value as shown in Figure 1.

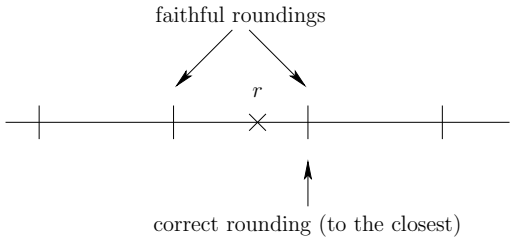

Figure 1: Faithful rounding

```
Faithful?(r:real,f:(Fbounded?(b))):bool =
   isMin?(b)(r,f) OR isMax?(b)(r,f)
```

This property is both easier to ensure than correct rounding and very powerful as it implies that the computed result and the exact result are very close. A faithful rounding also implies that the distance between the exact and the computed values is less than one ulp. The opposite is often assumed to be true. However, it does not hold when the floating-point number is a positive rounding of the radix. Figures 3 and 2 illustrate both situations. The darker strip corresponds to the set of reals r such that $|r - f| < \text{ulp}(f)$, the lighter strip corresponds to the set of real r such that f is a faithful rounding of r.

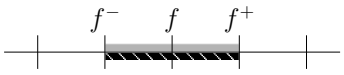

Figure 2: Ulp and faithful rounding

Figure 3: Ulp and faithful rounding when f is a positive power of the radix

Therefore, we have two different criteria to guarantee that a float is a faithful rounding of a real value (see [4] for more details). If $0 < f$ and $|f - r| < ulp(f^-)$ then f is a faithful rounding of r (`Faithful1`). If $0 < f$, $|f - r| < ulp(f)$ and $f \leq r$, then f is a faithful rounding of r (`Faithful2`).

```
Faithful1: Theorem
  Fcanonic?(b)(f) AND 0 < f AND
    abs(f-r) < Fulp(b)(Fpred(b)(f))
      => Faithful?(r,f)

Faithful2: Theorem
  Fcanonic?(b)(f) AND 0 < f AND
    abs(f-r) < Fulp(b)(f) AND f <= r
      => Faithful?(r,f)
```

4. POLYNOMIAL EVALUATION

In this section, we present an application of our formalization to polynomial evaluation using floating-point arithmetic. The basic ideas of this application were originally developed in Coq [4]. Due to the proof automation features provided by PVS, the results presented here are significantly better than the original ones.

Usually, when evaluating a polynomial by Horner's rule after an argument reduction, the last computation creates the most significant error in the final result. For example, for the computation of the exponential, we compute $1+x+x^2/2+\cdots$ assuming that $|x| \leq \ln(2)/2 \ll 1$. The errors in computing $x^2/2+\ldots$ are negligible compared to the final result whose value is about 1.

The basic step of a polynomial evaluation is the computation of expressions of form $a \times x + y$, where a, x and y represent approximations of real values a', x' and y'. In the general case, an exact rounding is impossible to guarantee. However, we show that a under certain hypotheses, a faithful rounding can still be obtained.

To compute $a \times x + y$, we first compute $t = \circ(a \times x)$ and, then, $u = \circ(t + y)$, where $\circ$ is the rounding to the nearest. We do not assume that the processor provides a fused multiply-and-add (FMA) operation that performs this computation with one rounding. The aim is to provide sufficient conditions on a, x, y, a', x', y' that guarantee that these computations are faithful.

4.1 Round-off Error

To use previous lemmas and theorems, such as `Faithful1`, we have to bound a floating-point number with the real values it rounds. The closer the bounds, the more general the sufficient conditions are for a faithful rounding.

If $f = \circ(r)$ is canonical and non-zero, then we prove that

$$\frac{|r|}{1+\frac{1}{2\times|n_f|}} \leq |f| \leq \frac{|r|}{1-\frac{1}{2\times|n_f|}}.$$

Note that the bounds above do not require f to be normal. If f is known to be normal, we can prove that

$$\frac{|r|}{1+\frac{\beta^{p-1}}{2}} \leq |f| \leq \frac{|r|}{1-\frac{\beta^{p-1}}{2}}.$$

```
RoundLe: Lemma
  Fcanonic?(b)(f) AND f /= 0 AND
  Nearest?(b)(r,f)
    => abs(f) <= abs(r)/(1-1/(2*abs(Fnum(f))))

RoundGe: Lemma
  Fcanonic?(b)(f) AND f /= 0 AND
  Nearest?(b)(r,f)
    => abs(r)/(1+1/(2*abs(Fnum(f)))) <= abs(f)
```

If the floating-point number is near a power of the radix, the ulp of its predecessor is twice smaller (see Figure 3). In this case, the previous lemmas cannot be applied. However, the rounding to the nearest is closer to the real value than its predecessor, and this distance can be expressed with the ulp of the predecessor.

```
NearestUlp2: Lemma
  Fcanonic?(b)(f) AND Nearest?(b)(r,f) AND
  abs(r) <= abs(f)+Fulp(b)(Fpred(b)(Fabs(f)))/2
    => abs(f-r) <= Fulp(b)(Fpred(b)(Fabs(f)))/2
```

These inequalities are very difficult to handle using a proof assistant: except for the last one, we have nested divisions! This implies that any computation involving these lemmas (and they will be thoroughly used in the next proofs) will need the proofs that both the values `2*abs(Fnum(f))` and `1+1/(2*abs(Fnum(f)))` are non-zero. This is easily handled in PVS as the division is defined only when the divisor is non-zero, using the predicate sub-typing feature. The PVS type-checker will try to automatically discharge these non-zero conditions every time a division is used and once for all. In contrast, in Coq the division is defined for all real numbers. However, all the interesting theorems have preconditions stating that the divisors are non-zero. This means that at several places in a formal proof, Coq will need a proof that `2*abs(Fnum(f))` is non-zero, and this proof will be asked many times. Even if the proof is not very difficult, it is tedious. It also increases the size of the proof script and of the proof object handled by the proof assistant. The solution we found in Coq was to put the fact that `0 < 2*abs(Fnum(f))` as a temporary lemma inside the proof. This property becomes a hypothesis that can be used everywhere within a proof, but it is proved only once at the end of the proof. This solution is not fully satisfactory:

- It is difficult to know in advance the lemmas that are required during the proof. Therefore, it is sometimes necessary to return to the beginning of the proof when a new hypothesis is required.
- The number of lemmas grows considerably as they are required for each divisor appearing in an expression. For large expressions, it means several uninteresting hypotheses that pollute the proof environment.

This problem is minimized in PVS via its predicate subtyping feature and its powerful type-checker.

4.2 Sufficient Conditions

From the results above, we can prove that the conditions

- $0 < u$,
- u is subnormal or $\beta \times |t| \leq u^-$, and
- $|y' - y + a' \times x' - a \times x| < \frac{\text{ulp}(u^-)}{4}$

are sufficient to guarantee that $u = \circ(y + \circ(a \times x))$ is a faithful rounding of the exact real value $a' \times x' + y'$.

```
AxpyPos: Lemma
  Nearest?(b)(a*x,t) AND Nearest?(b)(t+y,u) AND
  0 < u AND
  (Fnormal?(b)(t) => radix*abs(t) <= Fpred(b)(u))
  AND abs(y1-y+a1*x1-a*x)<Fulp(b)(Fpred(b)(u))/4
      => Faithful?(y1+a1*x1,u)
```

Unfortunately, we do not have a priori the outputs u and t to check if these conditions are satisfied. The following lemma provides conditions that can be checked *a priori* and without knowing the argument. If

- $p \geq 6$,
- $\left(\beta + 1 + \beta^{4-p}\right)|a \times x| \leq |y|$, and
- $|y' - y + a' \times x' - a \times x| < |y|\frac{\beta^{1-p}}{6\beta}$,

then $u = \circ(y + \circ(a \times x))$ is a faithful rounding of the exact real value $a' \times x' + y'$.

```
Axpy_opt: Theorem
  Nearest?(b)(a*x,t) AND Nearest?(b)(t+y,u) AND
  Prec(b) >= 6 AND
  (radix+1+radix^(4-Prec(b)))*abs(a*x)<=abs(y) AND
  abs(y1-y+a1*x1-a*x) <
  abs(y)*radix^(1-Prec(b))/(6*radix)
    => Faithful?(y1+a1*x1,u)
```

A particular case is when the radix is 2 and the precision is greater or equal to 24, i.e., IEEE single precision. In this case, if

- $3.000001\ |a \times x| \leq |y|$, and
- $|y' - y + a' \times x' - a \times x| < \frac{|y| \times 2^{1-p}}{12}$,

then $u = \circ(y + \circ(a \times x))$ is a faithful rounding of the exact real value $a' \times x' + y'$.

```
Axpy_simpl: Theorem
  Nearest?(b)(a*x,t) AND Nearest?(b)(t+y,u) AND
  Prec(b) >= 24 AND radix = 2 AND
  (3+1/100000)*abs(a*x) <= abs(y) AND
  abs(y1-y+a1*x1-a*x) < abs(y)*2^(1-Prec(b))/12
    => Faithful?(y1+a1*x1,u)
```

The advantages of PVS over Coq was striking in this application. The reason is that the ratio $\frac{\text{Computation}}{\text{Reasoning}}$ is very high here. On the contrary, for the proofs of Subsections 3.2.2 and 3.2.3, this ratio was low and the proofs were about as long and as difficult in PVS as in Coq. But here, the reasoning involved to prove that the polynomial evaluation is faithful is drowned into a deep sea of computations on real numbers involving many exponentiations. The consequence in Coq is a huge proof that is very difficult to read and nearly impossible to modify. On the other hand, the PVS proof was large, but the computations were more easily isolated from the reasoning and the global proof was more easily modified: the original proof was only radix-2, it was later upgraded into a generic-radix proof.

5. EXAMPLES

In this section, we apply the results presented in Section 4 to the evaluation of the Fike's polynomial and the computation of a Taylor series expansion of exponential.

5.1 Fike's Polynomial

The following polynomial is given in [10], where it is used as an approximation in IEEE single precision of 2^x over $[-1/16; 0]$. The coefficients after rounding are:

$$P(x) = 1 + \frac{1453635}{2097152}\,x + \frac{1007583}{4194304}\,x^2 + \frac{14899271}{268435456}\,x^3 + \frac{10327375}{1073741824}\,x^4 + \frac{11451013}{8589934592}\,x^5 + \frac{5179279}{34359738368}\,x^6.$$

The method error $|2^x - P(x)|/|x|$ is bounded by

$$\frac{8577801}{4503599627370496}.$$

It was proved in [4] that this computation is faithful over the interval $[-1/16; 0]$. Here, we can prove that this computation is faithful over the larger interval $[-1/4; 0]$.

5.2 Taylor Series Expansion of Exponential

Consider the Taylor expansion of the exponential, truncated at the degree n:

$$P_n(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots + \frac{x^n}{n!}.$$

To compute it, we round each coefficient to the nearest IEEE double precision number. We will therefore consider

$$\widetilde{P_n(x)} = 1 + x + \frac{x^2}{2} + \circ\left(\frac{1}{6}\right)x^3 + \cdots + \circ\left(\frac{1}{n!}\right)x^n.$$

The method error is $|\exp(x) - \widetilde{P_n(x)}|/|x|$. It decreases when n increases as P_n is a more accurate approximation of the exponential than P_m when $n > m$. It decreases when x decreases as the Taylor expansion was done for $x = 0$. The computation error is the difference between the result using Horner's rule $u = \circ(1 + x \times \circ(\frac{1}{2} + x \times \circ(\cdots)))$ and $\widetilde{P_n(x)}$.

Both errors contribute to the final error between u and $\exp(x)$. From the theorem `Axpy_simpl`, we find guaranteed intervals for x such that

- u is a faithful rounding of $\widetilde{P_n(x)}$, and
- u is a faithful rounding of $\exp(x)$,

for different values of n.

n	Faithful to $\widetilde{P_n(x)}$	Faithful to $\exp(x)$
2	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-27}$
3	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-18}$
4	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-13}$
5	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-10}$
6	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-8}$
7	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-6}$
8	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-5}$
9	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-4}$
10	$\lvert x\rvert \leq 2^{-3}$	$\lvert x\rvert \leq 2^{-4}$

The table shows the influence of the method error: the computation error is nearly always the same, we just need that $|x| \leq 2^{-3}$ to guarantee that u is a faithful rounding of $\widetilde{P_n(x)}$. When we add the method error, we have tighter bounds on x: when the degree of the polynomial decreases, the method error increases. The table tells us that to guarantee that the final result is a faithful rounding of $\exp(x)$, we have to limit the method error by tightly reducing the range of x.

6. CONCLUSION

We have implemented a fairly complete formalization of floating-point numbers in the verification system PVS based on a generic high-level specification written in the proof assistant Coq. On top of this formalization, we have developed, and formally proved, sufficient conditions that guarantee the accuracy of floating-point computations of polynomial evaluation. This work contains a total of 280 lemmas and theorems. Using PVS 3.2 on a 2.60GHz processor, it takes more than 20 minutes to check all the proofs.

Given the critical nature of some engineering applications, we believe that formal verification is a necessary step in the safety analysis of these digital systems. As many computations in numerical applications are polynomial evaluations or elementary function evaluations, the work presented here is fundamental to this analysis. In our experience, the combination of a high-level formalization and a powerful theorem prover will ease the formal verification effort. As tedious mathematical facts are automatically solved, the developer may focus on more abstract logical reasoning. The shifting

of the proofs of these facts from the user to the tool is of the uttermost importance for the more general use of formal methods in the numerical analysis community.

7. REFERENCES

[1] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[2] S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2004.

[3] S. Boldo and M. Daumas. Representable correcting terms for possibly underflowing floating point operations. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 79–86, Spain, 2003.

[4] S. Boldo and M. Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 37(1-4):45–60, 2004.

[5] W. J. Cody, R. Karpinski, et al. A proposed radix and word-length independent standard for floating point arithmetic. *IEEE Micro*, 4(4):86–100, 1984.

[6] T. Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal*, 20(4):129–135, 148, 1995.

[7] The Coq Development Team, INRIA. *The Coq Proof Assistant: Reference Manual, Version 8.0*, 2004.

[8] M. Daumas, L. Rideau, and L. Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

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

[10] C. T. Fike. Methods of evaluating polynomial approximations in function evaluation routines. *Communications of the ACM*, 10(3):175–178, 1967.

[11] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[12] J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[13] J. Harrison. Verifying the accuracy of polynomial approximations in HOL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 137–152, Murray Hill, New Jersey, 1997.

[14] J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics*, pages 113–130, Nice, France, 1999.

[15] J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 217–233, Austin, Texas, 2000.

[16] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. Second edition.

[17] C. Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, Computer Science Department, Saarland University, Saarbrucken, Germany, 2002.

[18] W. Kahan. Mathematics written in sand—the HP-15C, Intel 8087, etc. In *Statistical Computing Section, Proceedings of the American Statistical Association, Toronto*, pages 12–26, 1983.

[19] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[20] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

[21] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. C. Martin, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002.

[22] P. S. Miner and J. F. Leathrum. Verification of IEEE compliant subtractive division algorithms. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 64–78, 1996.

[23] J.-M. Muller. On the definition of ulp(x). Technical Report LIP RR2005-09 INRIA RR-5504, Laboratoire de l'Informatique du Parallélisme, 2005.

[24] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating point hardware. *Intel Technology Journal*, 3(1), 1999.

[25] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[26] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[27] D. M. Russinoff. A mechanically checked proof of correctness of the AMD-K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75, 1999.

[28] D. M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. *Lecture Notes in Computer Science*, 1954:3–36, 2000.

[29] P. H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.

[30] D. Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3):51–62, 1981.

[31] D. Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[32] Sun Microsystems. *Numerical Computation Guide*, 1996.