Generating formally certified bounds on values and round-off errors.
Marc Daumas, Guillaume Melquiond

To cite this version:
Marc Daumas, Guillaume Melquiond. Generating formally certified bounds on values and round-off errors.. [Research Report] LIP RR-2004-36, Laboratoire de l’informatique du parallélisme. 2004, 2+24p. hal-02102116

HAL Id: hal-02102116
https://hal-lara.archives-ouvertes.fr/hal-02102116
Submitted on 17 Apr 2019
Generating formally certified bounds on values and round-off errors

Marc Daumas and Guillaume Melquiond

Juillet 2004

Research Report N° 2004-36
Generating formally certified bounds on values and round-off errors

Marc Daumas and Guillaume Melquiond
Juillet 2004

Abstract

We present a new tool that generates bounds on the values and the round-off errors of programs using floating point operations. The tool is based on forward error analysis and interval arithmetic. The novelty of our tool is that it produces a formal proof of the bounds that can be checked independently using an automatic proof checker such as Coq and a complete model of floating point arithmetic. For the first time ever, we can easily certify that simple numerical programs such as the ones usually found in real time applications do not overflow and that round-off errors are below acceptable thresholds. Such level of quality should be compulsory on safety critical applications. As our tool is easy to handle, it could also be used for many pieces of software.

Keywords: Round-off error, Overflow, Formal proof, Certification, Safety critical

Résumé

Nous présentons un nouvel outil capable de générer, à partir de la description d’un programme qui utilise des opérations à virgule flottante, des bornes sur les valeurs des variables et les erreurs d’arrondi commises dans ce programme. L’outil est basé sur l’analyse d’erreur en mode direct et l’arithmétique d’intervalles. La nouveauté de notre outil est qu’il produit une preuve formelle des bornes qui peut être certifiée indépendamment en utilisant un outil automatique de preuve tel que Coq et un modèle complet de l’arithmétique à virgule flottante. Pour la première fois, il est possible de facilement certifier que les programmes numériques simples comme ceux habituellement utilisés dans les applications en temps réel ne commettent pas de dépassement de capacité et que les erreurs d’arrondi sont au-dessous de seuils acceptables. Un tel niveau de qualité devrait être obligatoire sur les applications critiques. Comme notre outil est facile à manipuler, ce niveau devrait également être utilisé pour de nombreux logiciels.

Mots-clés: Erreur d’arrondi, Dépassement de capacité, Preuve formelle, Certification, Logiciels critiques.
Generating formally certified bounds on values and
round-off errors*

Marc Daumas & Guillaume Melquiond

E-mail: Marc.Daumas@ENS-Lyon.fr & Guillaume.Melquiond@ENS-Lyon.fr

1 Introduction

The purpose of this work is to safely qualify the behavior of algorithms concerning the range of all the floating point variables and all the round-off errors. As this work was intended for safety critical applications, we generate a formal proof that is automatically verified by an automatic proof checker such as Coq [13]. However, as our tool is very easy to handle, it can be used for more or less safety critical applications, to provide formally certified bounds on the values and the round-off errors and guarantee that all variables and all round-off errors in a given program are under control.

Deadly and disastrous failures [14, 19, 9] have shown that common professional practices do not guarantee quality of the produced software. Pen-and-paper hand-written formal proofs are not sufficient as errors are commonly printed [17, 29] even in highly regarded publications by honored scientists in well reviewed and cited journals [27, 11]. Results produced by interval libraries, including for example [23, 15], are also not sufficient for safety critical applications. Some handwritten pen-and-paper proofs contain errors and there is no doubt that some highly regarded handwritten programs also contain errors even if they have been well reviewed and produced by honored scientists. In both cases, deriving the conclusions is not sufficient and the process should be certified by an automatic proof checker.

Such certifications have already been used to detect or prevent errors in algorithms and implementations [1]. Few results are available for computer arithmetic [20, 2] and even fewer results consider the effect of round-off errors in floating point arithmetic. Our contacts with leading teams at Intel and NASA LaRC allow us to safely believe that at the time we are reporting on this work, all developments that follow round-off errors are linked to one of the authors of this report [18, 4]. As certifying formal proofs by hand can be extremely tedious and needs specialized training, only carefully chosen applications have been developed so far.

Our tool has been developed to simplify such a task. Given a description of the application, it automatically generates proofs for the various bounds on values and round-off errors. As a result, it can be easily handled by naive users with no special training in Coq or any other formal proof verification tool. Our tool can be linked to many automatic proof checkers. The ones based on higher order logic such as Coq, PVS and HOL are probably the most conservative ones among them; Coq and HOL [10] are probably the safer ones as they rely solely on a very small kernel. Future development of our tool may let users check their proofs on two or three independent proof checkers and get a level of confidence higher than any guarantee provided so far.

It is visible in the following text that some decisions were still open for the implementation of an automatic tool such as ours. We propose educated choices that are useful in regard to common

---

*This material is based on work partially supported by National Aeronautics and Space Administration, Langley Research Center under the Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace. This text is also available as a research report of the Institut National de Recherche en Informatique et en Automatique http://www.inria.fr.
practices and to our first application that is used to maintain safety distances between aircrafts. Our main contribution is still our ability to produce forward error analysis proofs based on interval arithmetic that are certified by an automatic proof checker such as Coq as there is no other tool able to do it. Our choices will make a difference when comparable tools will appear in the future.

There might be many reasons for a computing system to fail. The first one might be that some hardware, the operating system or the compiler is not working correctly. Another one might be that the user did not handle the software correctly or that the designer did not understand exactly the intended behavior of the system. Yet another one might be that one programmed structure is not initialized or used correctly, and so on. All these problems are not new with floating point arithmetic and teams around the world are developing tools and methodologies to prevent them.

Failure may also be caused in the substitution of a mathematical question by a finite number of operations. For example when we truncate an infinite series or when we replace the mathematical solution of an equation by a constant number of steps of Newton-Raphson iterations [24, 26]. These errors are also handled elsewhere. Moreover, the initial mathematical question is not necessarily visible in the final compiled program. Indeed it is not always possible to get back to the initial algorithm knowing its implementation only.

Our tool bounds the gap between the behavior of the given program executed on real numbers, with no round-off error, and the behavior of the same program executed on a computer, with round-off errors. Although these bounds are very useful in practice, it is sometimes also interesting to bound the global effect of all errors on the computed values. This is possible with our tool by adding uncertainty to some constants and data of the program. The user is responsible to certify that the modified intervals appropriately account for all truncation and discretization errors of the program compared to the mathematical question. We did so for our aeronautic application and the additional error bounds will soon be formally certified with PVS.

2 Setting up a formal model adapted to automatic proof checkers

In the following, \( x : \mathbb{R} \) denotes a value \( x \) known with an infinite precision. This is a theoretical number and an algorithm uses an approximated representable value \( \bar{x} : \mathbb{F} \), with \( \mathbb{F} \) being the set of hardware-representable numbers. We do not mean that \( \bar{x} \) is the rounded value of \( x \), we mean that \( \bar{x} \) is the value that is used and possibly computed by the program in place of \( x \).

A closed mathematical function \( f(x) \) is implemented by a program \( \bar{f}(\bar{x}) \) that takes into account the round-off errors caused by the computation. Discretization and truncation errors have to be handled separately. We use \( \bar{x} \) instead of \( x \) to recall the reader that a program always uses only representable inputs.

The tool has been designed to automatically generate a proof of the range of \( \bar{f}(\bar{x}) \), the difference between the exact value \( f(x) \) and the computed value \( \bar{f}(\bar{x}) \), and so on, when some properties are known on the input variables. For example, under the hypothesis \( \bar{x} \in [1, 4] \), it is easy to prove that \( \sqrt{\bar{x}} \) lies in the range \([1, 2]\). The tool is here to help certifying proofs on complete applications.

The analysis is done on a low-level description of the application on which arithmetic properties have to be proved. When two variables are summed, our tools does not simply do an addition, it generates proofs about the implemented addition that a computer will effectively execute, for example a floating-point addition on simple precision numbers with respect to the IEEE-754 standard and its default rounding mode as defined in our formal specification. The description of the program must carry enough information in order for the proofs to relate to the real program as implemented and executed.

The tool also takes into account some properties (range, absolute error, relative error) on the input variables. Given the same kind of properties on the output variables, the tool tries to generate a script of proof that states and verifies these properties from the hypotheses on the input variables. Section 2.1 describes one of the mechanisms involved in proving range properties. Section 2.2 is devoted to the proofs of errors, both absolute and relative.
A proof-checker like Coq does not work on the actual proof but on a script of tactics that describes each necessary logic operation to validate the conclusion from the premises. The proof is generated by the automatic proof checker by playing the script. Coq kernel stores the proof as a lambda-terms following Curry-Howard isomorphism. The lambda-terms cannot be read by human beings but it can be double checked by tools independent from Coq kernel and Coq development teams. As it stands now, none of the files manipulated by Coq stores the proof in a human readable format. This is the reason why we always refer to the proof script that is played on the automatic proof checker to print on screen the steps of the proof.

2.1 Bounds on the computed values

Variables may be bounded into explicitly computed ranges certified by an automatic proof checker. For example, a responsible programmer should prove that inputs of all square root operations cannot be negative and that his program never divides by zero. Range checking also let us verify that no result will overflow as floating point algorithms that are not specially built to deal with them will often lead to unanticipated results if one or more operations overflow. Worse, an overflowing operation on non saturated two's complement fixed point arithmetic will return a wrapped value with the wrong sign.

Ranges are stored as pairs of numbers: a lower bound and an upper bound\(^1\). Such a representation is not optimal and authors may use other ones in the future. It is however sufficient in most cases and as a trade-off, we avoid manipulating complex structures. Indeed, formally checking a proof is a computation intensive operation and using simpler properties (a pair of numbers instead of the whole description of a domain) makes verifications easier.

What is interesting for a proof is the fundamental property of interval arithmetic: if the arguments of an expression are replaced by enclosing intervals, the expression evaluation will lead to an interval that contains the result. Such an inclusion property is exactly what is needed for a proof: given the ranges of the inputs, interval arithmetic will give the range of the output – or more generally an overestimation of this range, but it still is a compatible answer.

Our tool may have handled domains on variables \(x\) but our past experiences on programs lead us to rather bound computed values. The upper and lower bounds will be used in pre- and post-conditions of implemented functions. When a C program contains an assertion \((0.0 \leq x)\), \(x\) refers to the value actually in memory or in a register; consequently the assertion does not apply to \(x\) but to \(\hat{x}\). When a variable must be proved not to underflow or overflow, we mean to check the property on \(\hat{x}\), and so on. Most of the properties proved at this stage require knowing the domain of \(\hat{x}\); the domain of \(x\) may never be needed though it can be estimated with the error bound defined Section 2.2.

Our educated decision to bound the approximated variable rather than the exact quantities has another important consequence: there is no loss if the lower and upper bounds of \(\hat{x}\) are in \(\mathbb{F}\) as the approximated value \(\hat{x}\) is an element of \(\mathbb{F}\) by definition and we do not consider intervals of \(\mathbb{R}\) but of \(\mathbb{F}\). The consequence on formal proof checking is substantial, bounds on the exact values could need arbitrarily wide expressions to exactly represent real numbers. On the other hand, bounds on \(\hat{x}\) are rationals with a very limited precision (for example IEEE standard single or double precision numbers).

Common interval arithmetic considers convex subsets of \(\mathbb{R}\). Consequently, the implementation of the addition operator for a traditional interval arithmetic library is:

\[
[\hat{a}, \hat{b}] + [\hat{c}, \hat{d}] = [\bigvee (\hat{a} + \hat{c}), \bigtriangleup (\hat{b} + \hat{d})].
\]

There \(\bigvee\) and \(\bigtriangleup\) denote that operations should be rounded toward \(-\infty\) and \(+\infty\). The bounds may not carry enough precision for \(\hat{a} + \hat{c}\) and \(\hat{b} + \hat{d}\) to be exactly stored. It is the reason why the two operations are rounded so that the resulting interval encloses all possible values.

\(^{1}\)We will present in footnotes references to Coq definitions and properties included in annexes. The annexes have been generated through an automatic pretty-printer and they are faithful to the actual scripts automatically validated by Coq. The name of each annex section is the name of the file as it appears in our library. This first definition is given by \texttt{Einb\_bound} in the Coq support library.
In our cases, such rounding considerations are not necessary: the hardware-implemented addition ♦ is considered as a new exact operation with respect to $\mathbb{F}$ and not as the approximation of the addition on the field of the real numbers. That means that if
\[
\hat{x} \in [\hat{a}, \hat{b}] \quad \text{and} \quad \hat{y} \in [\hat{c}, \hat{d}] \quad \text{then} \quad x \oplus y \in [\hat{a} \oplus \hat{c}, \hat{b} \oplus \hat{d}]
\]
Indeed, the values $\hat{a} \oplus \hat{c}$ and $\hat{b} \oplus \hat{d}$ already lie in $\mathbb{F}$, there is no need for an additional rounding. Consequently, since the addition ♦ is a monotonous function of $\mathbb{F}^2 \rightarrow \mathbb{F}$ whatever the rounding mode, then $\hat{a} \oplus \hat{c}$ and $\hat{b} \oplus \hat{d}$ are the bounds.
\[
[\hat{a}, \hat{b}] \oplus [\hat{c}, \hat{d}] = [\hat{a} \oplus \hat{c}, \hat{b} \oplus \hat{d}].
\]
There is no need to take instead possibly different and non-optimal values $\land (\hat{a} + \hat{c})$ and $\triangleleft (\hat{b} + \hat{d})$.

Consequently, the most commonly used interval arithmetic libraries were not appropriate for our project. We chose the Boost parameterizable interval arithmetic library [3]. This is a C++ library that allows to specify the types and the arithmetic to be used instead of being limited to arithmetic on $\mathbb{R}$.

### 2.2 Bounds on the errors

The other arithmetic property that our tool proves is the difference between the exact result and the computed value. Finite precision of numeric computations is likely to induce a difference between the expected result and the computed value. Such differences will progressively build up and it is necessary for the user to certify that the computed result is meaningful.

Bounds on values and errors are quite similar and we could have decided to handle them at the same time. Each variable would have been represented by four numbers (two ranges) and there would have been new operators inspired by interval arithmetic that would compute on these quartets. Such a representation would have given shorter proof scripts. However there are situations where such a simplicity would be restrictive.

For example, the error range for the expression $x/\sqrt{1 + x^2}$ would be easily computed by this method, but the domain of the result would not. Indeed the result is in the range $[-1, 1]$ for any $x$ [16], and this tight interval may be a precondition for another part of the algorithm. Yet, interval arithmetic is usually not able to yield such a small interval: multiple occurrence of the same variable in the expression leads to an overestimation of the result usually called decorrelation.

Interval arithmetic is unable to keep track of the relations between sub-expressions and $x/\sqrt{1 + x^2}$ is seen as if it was $x/\sqrt{1 + y^2}$, with $y$ having the same domain as $x$ but not necessarily the same value. Consequently, in this example, if $x$ is in the range $[0, 2]$, $\sqrt{1 + x^2}$ is in the range $[1, 0] \circ (\sqrt{y})$. Standard interval arithmetic will then answer that the whole expression is in $[0, 2] \div [1, 0] \circ (\sqrt{5}) = [0, 2]$. This interval is no subset of the range $[-1, 1]$ as another part of the algorithm could expect. Standard interval arithmetic cannot be used to validate the domain of this expression and human interaction is needed instead.

Consequently, it may be preferable for bounds and errors proofs to be separated so that if our tool is not able to prove one of them it may still be used for the other one limiting human interactions as much as possible.

#### 2.2.1 Absolute error

The most commonly encountered kind of error representations are absolute and relative errors [7]. Absolute error is useful with fixed point arithmetic since the magnitude of the rounding error does not depend on the magnitude of the computed value. It may also be useful with floating point arithmetic when a subtraction cancels, when the exponents of the results stay close or with subnormal numbers (including zero). Relative errors can be used in all the other cases.

---

2This result is given by $\text{Eplus\_Rec\_bound}$ for generic floating point numbers.
Generating formally certified bounds on values and round-off errors

Absolute error\(^3\) \(\eta_x\) is attached to variable \(x\) by equality

\[
x = \tilde{x} + \eta_x.
\]

We will not define the behavior of all the implemented operators as forward error analysis is very common. As an example, here is the formula giving the absolute error for the addition. \(\eta_0\) is the rounding error between the exact result of \(\tilde{x} + \tilde{y}\) and its computed value \(\tilde{x} \oplus \tilde{y}\).

\[
z - \tilde{z} = (x + y) - (\tilde{x} \oplus \tilde{y}) = (x + y) - (\tilde{x} + \tilde{y} - \eta_0) = (x - \tilde{x}) + (y - \tilde{y}) + \eta_0 = \eta_x + \eta_y + \eta_0.
\]

We extend the definition to interval bounds. Intervals \(A_x\) and \(A_y\) are attached to variables \(x\) and \(y\) and respectively enclose \(\eta_x\) and \(\eta_y\). Interval \(A_0\) encloses all possible values for \(\eta_0\). This interval is given by the properties of the underlying arithmetic. For fixed point arithmetic, it is a constant interval. For floating point arithmetic, it depends on the value of \(\tilde{z}\). Our tool detects automatically Sterbenz’s conditions and proves that \(A_0\) is equal to zero if the theorem is verified for all \(\tilde{x}\) and \(\tilde{y}\) [28, 3].

\[
\eta_x \in A_x \subseteq A_x + A_y + A_0
\]

Bounds on values of Section 2.1 were not real numbers and the precision of interval arithmetic was very limited. It is not the case for error intervals and we used common interval arithmetic with rounded overestimations.

2.2.2 Relative error

We only consider relative error\(^5\) when \(x\) and \(\tilde{x}\) have the same sign. The most common representation of this error is \(\tilde{x} = x \times (1 + \epsilon_1)\) but another possible one is \(x = \tilde{x} \times (1 + \epsilon_2)\) [12].

In order to select one of these two representations, let us consider the case where \(\tilde{x}\) is one representable number closest to \(x\) (there is usually only one \(\tilde{x}\) value unless \(x\) is a mid-point between two representable values). Provided we are sufficiently far from the underflow threshold,

\[
\frac{u}{2} \leq -\frac{u}{2 + u} \leq \epsilon_1 \leq \frac{u}{2 + u} \leq \frac{u}{2},
\]

and

\[
\frac{u}{2} \leq \frac{2}{1 + u} \leq \epsilon_2 \leq \frac{u}{2} \leq \frac{u}{2},
\]

where \(u\) is the difference between 1 and the next representable floating point number. This bound is important because it is used after each operation to incorporate the additional rounding error.

Now if \(\tilde{x}\) is the smallest positive normalized number,

\[
\frac{u}{2} \leq -\frac{u}{2 + u} \leq \epsilon_1 \leq \frac{u}{2 - u} \leq \frac{u}{2},
\]

and

\[
\frac{u}{2} \leq \frac{u}{2} \leq \epsilon_2 \leq \frac{u}{2} \leq \frac{u}{2}.
\]

As a conclusion using \(x = \tilde{x} \times (1 + \epsilon_x)\) gives a range valid on all normalized numbers and consequently only the exponent of \(\tilde{x}\) needs to be checked. Once again, we provide only one

\[^3\text{This definition is given in Coq by } \texttt{Eint\_absol}, \text{ and by } \texttt{Eint\_error} \text{ for symmetric interval.}\]

\[^4\text{This result is given in the general case by } \texttt{Eplus\_error} \text{ and for floating point numbers by } \texttt{Eplus\_flec\_error} \text{ and } \texttt{Eplus\_exact\_flec\_error} \text{ depending on whether Sterbenz’s conditions on exponents are verified.}\]

\[^5\text{This definition is given by } \texttt{Eint\_relat}.\]
example as forward error analysis is very common. For the multiplication $z = x \times y$, if $\tilde{z}$ is always normalized, the relative error\footnote{This result is given by \texttt{Emult-relat} in the general case.} is:

\[
\frac{\tilde{z}}{z} = \frac{x \times y}{\tilde{x} \times \tilde{y}} = \frac{x \times y}{x \times y} (1 + \epsilon_0) = (1 + \epsilon_x)(1 + \epsilon_y)(1 + \epsilon_0).
\]

Once again, we attach intervals $R_x$ and $R_y$ to respective variables $x$ and $y$ enclosing $\epsilon_x$ and $\epsilon_y$. Let interval $R_0$ enclose the value of the relative error $\epsilon_0$. Its range depends on the properties of the arithmetic and we have seen that in the common cases of floating point arithmetic rounding to nearest, it is $[-u/2, u/2]$ and finally

\[
\epsilon_z \in R_z \subseteq (1 + R_x)(1 + R_y)(1 + R_0) - 1.
\]

All the other cases are handled in our library. By using automatic proof checkers, we guarantee that no case has been forgotten or inappropriately handled.

\subsection{Combining absolute and relative errors}

There may be cases where both an interval $A$ for the absolute error and an interval $R$ for the relative error can be computed and certified for a given variable $x$. Since the two intervals are linked by $\eta_x = \tilde{x}\epsilon_x$, if $I$ is an interval enclosing $\tilde{x}$, then $A / I$ is also an interval that encloses the relative error, and $R \times I$ encloses the absolute error. So starting from $A$ and $R$, we can build two sequences of intervals.

\[
\begin{cases}
A_0 = A \\ R_0 = R
\end{cases}
\quad \text{and} \quad
\begin{cases}
A_{n+1} = A_n \cap (R_n \times I) \\ R_{n+1} = R_n \cap (A_n / I)
\end{cases}
\]

Such a result would not be useful if a lot of iterations were necessary, since the proof script would become huge. Fortunately, it can be proved that the sequences are stationary after $A_1$ and $R_1$. So, if both $A$ and $R$ are available, possibly tighter intervals can easily be computed.

\subsection{Formal proof checking}

Our tool generates a Coq script of the proof. Such a proof script does not live by itself; it only describes what elementary facts should be applied, it does not provide the proofs of these theorems. Indeed these facts are provided by various libraries: theorems on real numbers arithmetic (the Coq standard library), specification and basic properties of floating point arithmetic [6]. It also depends on a library about interval arithmetic and its floating point extensions we developed so that each generated proof could be formally verified. For example, in order to use interval arithmetic for the addition, this composition property is needed:

\[
\tilde{a} \oplus \tilde{c} \leq \tilde{b} \oplus \tilde{d} \quad \text{if} \quad \tilde{a} \leq \tilde{b} \quad \text{and} \quad \tilde{c} \leq \tilde{d}.
\]

Non trivial theorems like Sterbenz's one are also stated [3], extended to floating point intervals in our library, and used by our tool.

On the other hand, our tool is not a part of Coq program and it can also be used with other provers such as HOL Light or PVS with ProofL[[1]]er package that supports batch proving and proof scripting in PVS. To preserve the independence of our tool, we do not use any technique specific to one prover such as tactics and automatic proof developments. Adapting our tool to other provers.

\footnote{\url{http://research.nlanr.org/~munoz/ProofLite/}.}
will only be a matter of syntax as soon as libraries for computer arithmetic are available on any alternate prover.

We benefit from the fact our tool is not a part of any prover and consequently that it does not need to be proved. For example, when proving the property \( \hat{a} \leq \hat{b} \oplus \hat{c} \), the tools acts as an oracle and computes beforehand the sum \( \hat{d} = \hat{b} \oplus \hat{c} \) and provides this number \( \hat{d} \) to the proof checker. Consequently, the checker does not have to explicitly compute \( \hat{b} \oplus \hat{c} \) in order to prove the property, it only has to verify that \( \hat{d} \) as provided by the tool is the closest\(^8\) (with respect to the rounding mode of \( \oplus \)) floating point number to \( \hat{b} + \hat{c} \), and then that \( \hat{a} \leq \hat{d} \). Both operations can be done by the existing Coq floating point library. Using the tool as an oracle avoids having to implement in the checker and prove all the floating point operations with their various rounding.

This solution works fine for the addition and the multiplication as such operations are internal on dyadic numbers and they are provided by the library. Dyadic numbers are pairs \((m, e)\) of \(\mathbb{Z}^2\) interpreted as \(m \times 2^e\). Radix 2 floating point numbers are dyadic numbers and our library relies on these. However mathematical division and square root of dyadic numbers do not generally produce dyadic numbers. Consequently these operators on real numbers cannot be directly used on dyadic numbers and the proofs cannot rely on them. Fortunately these operations can be reverted when rounding to nearest. For example,

\[
\hat{c} = \sigma(\hat{a} \div \hat{b}) \text{ iff } \hat{b} \times \hat{c}^+ \text{ is closer from } \hat{a} \text{ than } \hat{b} \times \hat{c}^- \text{ and } \hat{b} \times \hat{c}^+
\]

where \(\hat{c}^-\) and \(\hat{c}^+\) are the floating point numbers respectively just below and just above \(\hat{c}\).

Error bounds are also represented on dyadic numbers from the library. Our decision certainly reduces precision, but the situation is sensible in regard to common practices and it is also fully parameterizable. Moreover, we decided to further limit precision of the error bounds. Since our tool is much faster than any automatic proof checker, it tries to simplify the numbers used in the proof as much as possible. Without any simplification, the length of the error bounds grows continually. So does the time needed to verify each statement of the script as the proof goes on.

For example, we may replace \(m \times 2^e\) by \([m \cdot 2^{-k}] \times 2^{e+k}\) for a given \(k\). The bound is larger but certifying it is faster as \([m \cdot 2^{-k}]\) is a smaller constant. Our tool enlarges all the error ranges as long as they remain smaller than the ones the tool tries to prove. It is an efficient way to reduce the script size and the certification time since the main part of an error is generally its magnitude, not its exact value. For example, certifying the simplified proof was four times faster on the application of Section 3.

### 3 Maintaining safety distances between aircrafts

There is a lot of work done on shifting responsibility for aircraft safety distances from air traffic controllers to pilots themselves. This involves providing pilots with conflict detection systems. One algorithm \([22]\) works in an Euclidean three-dimensional neighborhood and a translation is necessary from geodetic coordinates as given by positioning systems \([21]\) to Euclidean coordinates.

Airplanes are protected by a cylinder (typically 5 miles diameter and 1000 feet height) and a conflict is defined as the overlapping of two protected zones. The above mentioned work has certified that it is equivalent to detect conflicts relative to the intruder. In this case, the intruder has a protected zone of radius 5 miles and height of 2000 feet. The ownership is just a point. A conflict is then defined as the incursion of the ownership in the protected zone of the intruder. The purpose of conflict detection is to detect such intrusions ahead of time and to propose escape and recovery trajectories.

A lot of approximations are necessary to solve this problem: position and speed of the planes are not exactly known, the use of an Euclidean neighborhood instead of geodetic coordinates creates modeling errors, translation itself incurs modeling and round-off errors. All these errors need to be taken into account so that the safety of the systems is guaranteed. The cylinder is

---

\(^8\)This property is verified by the boolean function \texttt{evaluate\_even\_closest} for rounding to nearest and tie-breaking to even.
easily inflated so that fewer and fewer trajectories may cross the initial safety cylinders. Yet, it is life critical to guarantee that the cylinder has been sufficiently inflated to remove any incorrect behavior.

As mentioned, there already is a published algorithm that works in an Euclidean three-dimensional neighborhood. We will describe here a new algorithm responsible of the translation part and use it as an example of safety critical application that can be formally proved thanks to our tool.

The inputs of the Algorithm 1 are the longitudes $\lambda_{\{1,2\}}$ and the geodetic latitudes $\phi_{\{1,2\}}$ of planes 1 and 2, and their speeds toward north $v_{N_{\{1,2\}}}$, toward east $v_{E_{\{1,2\}}}$ The outputs are the relative coordinates and the speed of both planes in an Euclidean neighborhood. $t_r$ is a constant. $R_m$ and $R_p$ are Earth radius along a meridian and along a parallel for a given latitude:

$$R_p(\phi) = \frac{a}{1 + (1 - f)^2 \tan^2 \phi}$$
$$R_m(\phi) = \frac{a(1 - e^2)}{(1 - e^2 \sin^2 \phi)^{3/2}}$$

Constants $a$, $e$ and $f$ are described by WGS84\(^9\) and are also used to define the geodetic coordinates in the positioning systems.

**Algorithm 1** Translation from geodetic coordinates to an Euclidean neighborhood

**Input:** $\phi_1$, $\phi_2$, $\lambda_1$, $\lambda_2$, $v_{N1}$, $v_{N2}$, $v_{E1}$, $v_{E2}$

$\phi_0 \leftarrow (\phi_1 + \phi_2)/2$
$r_{p0} \leftarrow R_p(\phi_0)$
$s_{N1} \leftarrow v_{N1}/R_m(\phi_1)$
$s_{N2} \leftarrow v_{N2}/R_m(\phi_2)$
$s_x \leftarrow (\lambda_1 - \lambda_2) * r_{p0}$
$s_y \leftarrow (\phi_1 - \phi_2) * R_m(\phi_0)$
$v_{x1} \leftarrow v_{E1} * r_{p0}/R_p(\phi_1 + s_{N1} * t_r)$
$v_{x2} \leftarrow v_{E2} * r_{p0}/R_p(\phi_2 + s_{N2} * t_r)$

These functions use trigonometric functions that must be approximated on computers. Therefore, we decided to approximate directly $R_m$ and $R_p$. This adds more errors as the algorithm uses

\(^9\)[http://www.wgs84.com/](http://www.wgs84.com/).
polynomials $\hat{R}_m$ and $\hat{R}_p$ rather than $R_m$ and $R_p$. The program for $\hat{R}_p$ is given Algorithm 2. $\hat{R}_m$ is omitted but it is simple implementation of Horner’s rule. We introduce an uncertainty on the first coefficient of each polynomial to account for the difference between the exact function and the approximated polynomial in our tool.

**Algorithm 2 Definition of $R_p(\phi)$ approximating $R_p(\phi)$**

**Input:** $\phi \{\text{all the dyadic numbers are floating point numbers}\}$

\[
x \leftarrow 511225 \times 2^{-18} - \phi^2 \\
y \leftarrow 4439091 \cdot 2^{-2} + x \times (9023647 \cdot 2^{-2} + x \times (13868737 \cdot 2^{-6} + x \times (13233647 \cdot 2^{-11} + x \times (-1898597 \cdot 2^{-14} + x \times (-6661427 \cdot 2^{-17})))))
\]

The accuracy of the results depends on the precision of the arithmetic. For this example, we used underlying IEEE-754 simple precision\(^{10}\) floating point arithmetic with rounding to nearest (with even tie-breaking) and subnormal numbers are implemented. The speed corresponds to an airliner and the latitudes range from 30 °N to 40 °N.

Our tool produced a 1650-lines long script. The generated proof guarantees that the absolute error on $s_z$ and $s_y$ is bounded by half a meter, and the error on $v_{x1}$ and $v_{x2}$ is bounded by $5 \times 10^{-4}$ m/s. We conclude that the safety cylinder radius should be increased by 2.60 m for a 300 seconds window.

The 1650 lines of the script can be decomposed between 850 lines of definitions and 800 lines of theorems and proofs. Only these 800 last lines are important with respect to computer arithmetic. The huge number of definitions is caused by the need to describe each operation and each number. For example, when proving a variable is in a given interval, the lower and upper bounds need to be defined separately first.

On the test-machine, the script is verified by Coq in 45 seconds. The verification is a lot slower than the generation itself (a split second). This gap is due to the way Coq handles integer arithmetic: integers are stored as lists of bits and the operations are computed one bit at a time (since it is the way integer arithmetic is proved in Coq standard library).

4 Conclusion and Perspectives

The tool that we have just presented is very limited. Yet it is complete enough to deal with our airplane safety distance example. The tool generates a script that can be formally verified by Coq and that proves bounds on the absolute error of the outputs. It can also easily generate a script for the case where the inputs are only known with a given limited accuracy.

Adding functionalities to our tool is time consuming. We have to state and prove the supporting theorems in our library. Today, our tool is limited to floating point arithmetic (rounded to nearest) and absolute error bounds. As there is no difficult lock, more and more functionalities will be incorporated such as fixed point arithmetic and additional arithmetic operations. We may also take into account some DSP specific implementation of floating point arithmetic. Yet it is always possible to state quickly a few unvalidated axioms to incorporate one specific function for setting up a demonstration.

Another perspective is a connexion to Why tool [8]. Why treats C and ML programs and generates proof obligations for automatic proof checkers. As no tool was available for floating point arithmetic, variables were treated as real numbers with no rounding error. We will cooperate with Why developers to extend it with the existing floating point arithmetic.

Finally, our tool only performs simple forward error analysis. It may be linked with other tools such as Fluctuat [25] that is able to propose contracting intervals for loops. Only the certification

\(^{10}\)The specialization of generic floating point properties to this format is in the Coq module Efloats754iec.
step has to be validated by an automatic proof checker for the result to be fully guaranteed. One may use any tool to provide useful oracles as long as oracles are finally validated by the proofs produced by our tool.

5 Acknowledgment

The authors would like to thank César Muñoz from the National Institute of Aerospace and Víctor Carreño from the NASA Langley Research Center, who raised our interest on the critical application used as an example, for their guidance and discussions on aeronautical matters.

References


Module ClosestCalc

Add LoadPath "Float".
Require AllFloat.

Section ClosestCalc.
Variable ndix: Z.
Variable bound: Float.
Variable precision: nat.

Coercion Local FloRadix := (FloR radix).
Hypothesis ndixMoreThanOne := (Zlt (POS xH) ndix).

Local radixNotZero := (Zlt (POS xO) ? (Zlt x weak ? ? ndixMoreThanOne)).
Hypothesis precisionNotZero := ¬ precision = 0.
Hypothesis precisionMoreThanOne := (lt (1) precision).
Hypothesis pGivesBound : (POS (vNum bound)) = (Zpower_nat ndix precision).

Inductive validity: Set :=
| bad: validity
| tie: validity
| good: validity.

Local dp := [x,r:float] [f:float->float]
(Fminus radix x (f (FPred bound radix r))).
Local dd := [x,r:float] [f:float->float]
(Fabs (Fminus radix (f r x))).
Local ds := [x,r:float] [f:float->float]
(Fminus radix (f (FSucc bound radix precision r)) x).

Definition evaluate_rounding := [x,r:float] [f:float->float]
let dd = (dd x r f) in
Cases (Fcompare ndix dd (dp x r f)) of
| INFERIEUR ⇒ bad
| EGAL ⇒ Cases (Fcompare ndix dd (ds x r f)) of
| INFERIEUR ⇒ bad
| EGAL ⇒ tie
| SUPERIEUR ⇒ tie
end
| SUPERIEUR ⇒
Cases (Fcompare ndix dd (ds x r f)) of
| INFERIEUR ⇒ bad
| EGAL ⇒ tie
| SUPERIEUR ⇒ good
end.

Definition evaluate_closest := [x,r:float]
Cases (evaluate_rounding x r [y:float|y]) of
| bad ⇒ false
| _ ⇒ true
end.

Lemma Rminus_le_r: (r1,r2:R) "0 ≤ r2: r1" → "r1 ≤ r2".

Theorem evaluate_closest_correct: (x,r:float)
(evaluate_closest x r) = true → (Fcanonic radix bound r) →
(Closest bound radix x r).

Definition is_positive_even := [x:positive]
Cases x of
| (xO _) ⇒ true
| _ ⇒ false 
| end.

Definition is_float_even := [x:float] 
Cases (Fnum x) of 
| (POS y) ⇒ (is_positive_even y) 
| (NEG y) ⇒ (is_positive_even y) 
| ZERO ⇒ true 
end.

Lemma is_float_even_correct: (x:float) 
(Fmonic radix bound x) → (is_float_even x) → true → (FEven bound radix precision x).

Definition evaluate_even_closest := [x,r:float] 
Cases (evaluate_arounding x r [y:float] y) of 
| bad ⇒ false 
| tie ⇒ (is_float_even r) 
| _ ⇒ true 
end.

Theorem evaluate_even_closest_correct: (x,r:float) 
(evaluate_even_closest x r) → true → (Fmonic radix bound r) → 
(EvenClosest bound radix precision x r).

End ClosestCalc.

Module Ebase

Require Reals.

Section Ebase.

Variable base: Set.
Variable base_plus: base → base → base → Prop.
Variable base_minus: base → base → base → Prop.
Variable base_mult: base → base → base → Prop.
Variable base_coeq: base → R.

Coercion Local base2R := base_coeq.

Definition base_plus := [a,b,c:base] (x,y,z:base) 
"a ≤ x" → "b ≤ y" → (base_plus x y z) → "c ≤ z".

Definition base_plus += [a,b,c:base] (x,y,z:base) 
"a ≤ x" → "y ≤ b" → (base_plus x y z) → "c ≤ z".

Definition base_minus := [a,b,c:base] (x,y,z:base) 
"a ≤ x" → "y ≤ b" → (base_minus x y z) → "c ≤ z".

Definition base_minus += [a,b,c:base] (x,y,z:base) 
"a ≤ x" → "b ≤ y" → (base_minus x y z) → "c ≤ z".

Definition base_mult := [a,b,c,d,e,f:base] (x,y,z:base) 
"a ≤ x" → "c ≤ e" → "b ≤ y" → "d ≤ f" → 
(base_mult x y z) → "e ≤ z ≤ f".

Variable error: Set.
Variable error_coeq: error → R.

Coercion Local error2R := error_coeq.

Definition base_plus_e := [a,b,c,d:base] [e:error] (x,y,z:base) 
"a ≤ x" → "c ≤ e" → "b ≤ y" → "d ≤ e" → 
(base_plus x y z) → "(Rabsolu (x+y-z)) ≤ e".

Definition base_minus_e := [a,b,c,d:base] [e:error] (x,y,z:base)
"a \leq x^\mu \rightarrow "x \leq c^\mu \rightarrow "b \leq y^\mu \rightarrow "y \leq d^\mu \rightarrow
(base\_minus\_x y z) \rightarrow (Rabsol (x-y - z)) \leq e^\mu.

Definition base\_mul\_c := [a, b, c, d: base] \[ c: error \] (x, y, z: base)
"a \leq x^\mu \rightarrow "x \leq c^\mu \rightarrow "b \leq y^\mu \rightarrow "y \leq d^\mu \rightarrow
(base\_mul\_x y z) \rightarrow (Rabsol (x * y - z)) \leq e^\mu.

Definition Eint\_bound := [a, dx: base] \[ x: base \] "xl \leq xr \leq xu^\mu.
Definition Eint\_error := [xe: error] \[ x: base \] \[ xx: R \]
"(Rabsol (xx \times xe - xx)) \leq x x e^\mu.

Lemma Eint\_error\_trans:
(me, xe: error) \[ xx: base \] \[ xx: R \]
(Eint\_error me xx xe xx) \rightarrow (Rle me xe) \rightarrow (Eint\_error xx xe xx).

Definition Eint := [dx: base] \[ xe: error \] \[ x: base \] \[ xx: R \]
(Eint\_bound xl xu x) \land (Eint\_error xe xx xx).

Lemma Eplus\_error\_aux:
(xe, ye: error) \[ xx: base \] \[ xx: R \]
(Eint\_error xe xx xe xx) \rightarrow (Eint\_error ye yy yy) \rightarrow
"(Rabsol ((xx + yy) - (xx + yy))) \leq xe + ye e^\mu.

Lemma Eplus\_error:
(xe, ye, ze, x: error) \[ xx: base \] \[ xx: R \]
(Eint\_error xe xx xe xx) \rightarrow (Eint\_error ye yy yy) \rightarrow
(base\_plus xx xx xx) \rightarrow
"xe + ye + ze1 \leq ze^\mu \rightarrow "(Rabsol (xx + yy - xx)) \leq ze1 e^\mu \rightarrow
(Eint\_error xe xx xx + yy).

Lemma Eminus\_error\_aux:
(xe, ye: error) \[ xx: base \] \[ xx: R \]
(Eint\_error xe xx xe xx) \rightarrow (Eint\_error ye yy yy) \rightarrow
"(Rabsol ((xx-y) - (xx-y))) \leq xe + ye e^\mu.

Lemma Eminus\_error:
(xe, ye, ze, x: error) \[ xx: base \] \[ xx: R \]
(Eint\_error xe xx xe xx) \rightarrow (Eint\_error ye yy yy) \rightarrow
(base\_minus xx xx xx) \rightarrow
"xe + ye + ze1 \leq ze^\mu \rightarrow "(Rabsol (xx-yy - xx)) \leq ze1 e^\mu \rightarrow
(Eint\_error xe xx xx).

Lemma Emul\_error\_aux:
(xe, ye: error) \[ xx: base \] \[ xx: R \]
(Eint\_error xe xx xe xx) \rightarrow (Eint\_error ye yy yy) \rightarrow
"Rabsol xx \leq xe^\mu \rightarrow "(Rabsol yy \leq ye^\mu \rightarrow
"Rabsol (xx \times yy - xx \times yy)) \leq xx \times xe + ye \times xe + xe \times ye e^\mu.

Lemma Emul\_error:
(xe, ye, ze, x: error) \[ xx: base \] \[ xx: R \]
(Eint\_error xe xx xe xx) \rightarrow (Eint\_error ye yy yy) \rightarrow
(base\_mul xx xx xx) \rightarrow "Rabsol (xx \times xx - xx) \leq ze1 e^\mu \rightarrow
"Rabsol xx \leq xe^\mu \rightarrow "(Rabsol yy \leq ye^\mu \rightarrow
"xe \times ye + ye \times xe + xe \times ye + xe \times ye \leq ze^\mu \rightarrow
(Eint\_error xe xx xx + ye).

End Ebase.

**Module Ebase2**

Require Import Reals.
Require Import BaseR.

Section Ebase2.
Variable base : Set.
Variable base_plus : base → base → base → Prop.
Variable base_minus : base → base → base → Prop.
Variable base_mult : base → base → base → Prop.
Variable base_coerc : base → R.

Coercion Local base2R := base_coerc.

Definition base_plus (a b c : base) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (a \leq x) \% R \rightarrow (b \leq y) \% R \rightarrow \text{base\_plus} \, x \, y \, z \rightarrow (c \leq z) \% R. \)

Definition base_plus_a (a b c : base) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (x \leq a) \% R \rightarrow (y \leq b) \% R \rightarrow \text{base\_plus} \, x \, y \, z \rightarrow (z \leq c) \% R. \)

Definition base_minus (a b c : base) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (a \leq x) \% R \rightarrow (y \leq b) \% R \rightarrow \text{base\_minus} \, x \, y \, z \rightarrow (c \leq z) \% R. \)

Definition base_minus_a (a b c : base) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (x \leq a) \% R \rightarrow (y \leq b) \% R \rightarrow \text{base\_minus} \, x \, y \, z \rightarrow (z \leq c) \% R. \)

Definition base_mult (a b c d f : base) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (a \leq x) \% R \rightarrow (c \leq e) \% R \rightarrow (b \leq y) \% R \rightarrow (d \leq f) \% R \rightarrow \text{base\_mult} \, x \, y \, z \rightarrow (e \leq z \leq f) \% R. \)

Variable error : Set.
Variable error_coerc : error → R.

Coercion Local error2R := error_coerc.

Definition base_plus_e (a b c d : base) (e : error) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (a \leq x) \% R \rightarrow (x \leq e) \% R \rightarrow (b \leq y) \% R \rightarrow (d \leq e) \% R \rightarrow \text{base\_plus} \, x \, y \, z \rightarrow (Rabs \, (x + y - z) \leq e) \% R. \)

Definition base_minus_e (a b c d : base) (e : error) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (a \leq x) \% R \rightarrow (x \leq e) \% R \rightarrow (b \leq y) \% R \rightarrow (d \leq e) \% R \rightarrow \text{base\_minus} \, x \, y \, z \rightarrow (Rabs \, (x - y - z) \leq e) \% R. \)

Definition base_mult_e (a b c d : base) (e : error) :=
  \( \forall \, x \, y \, z : \text{base}, \)
  \( (a \leq x) \% R \rightarrow (x \leq e) \% R \rightarrow (b \leq y) \% R \rightarrow (d \leq e) \% R \rightarrow \text{base\_mult} \, x \, y \, z \rightarrow (Rabs \, (x \times y - z) \leq e) \% R. \)

Definition Eint_bound_R (x l x u x r : R) := \( (x l \leq x u) \% R. \)
Definition Eint_absol_R (x e x f x r x x : R) :=
  \( (x e \leq x f - x r \leq x f) \% R. \)
Definition Eint_lat_R (x e x f x r x x : R) :=
  \( x r \neq 0 \% R \Rightarrow R \land (0 < 1 + x e) \% R \land \{1 + x e \leq x r / x r \leq 1 + x f\} \% R. \)
Definition Eint_bound (x l x u x r : base) := \( (Eint\_bound_R \, x l \, x u \, x r). \)
Definition Eint_absol (x e x f : error) (x r : base) (x x : R) :=
  \( (Eint\_absol_R \, x e \, x f \, x r \, x x). \)
Definition Eint_lat (x e x f : error) (x r : base) (x x : R) :=
  \( (Eint\_lat_R \, x e \, x f \, x r \, x x). \)

Lemma Eplus_absol_R :
∀ (xe xf ye yf xr yr xx yy : R),
Ein.absol_R xe xf xx xx →
Ein.absol_R ye yf yr yy →
Ein.absol_R (xe + ye) %R (xf + yf) %R (xr + yr) %R (xx + yy) %R.

Lemma Ein.absol_R:
∀ (xe xf ye yf xe xf ye yf : error) (xr yr xe : base) (xx yy : R),
Ein.absol_R xe xf xx xx → Ein.absol_R ye yf yr yy →
base_plus xe yr xe →
(ze ≤ xe + ye + we) %R → (xf + yf + wf ≤ zf) %R →
Ein.absol_R we wf xr (xy + yr) %R →
Ein.absol_R xe xf (xx + yy).

Lemma Ein.rel_R:
∀ (xe xf ye yf xe xf ye yf : error) (xr yr xe : base) (xx yy : R),
Ein.rel_R xe xf xx xx → Ein.rel_R ye yf yr yy →
base_plus xe yr xe → (base_coef xe) ≠ 0 %R → (0 < 1 + xe) %R →
(ze ≤ (1 + xe) × (1 + ye) × (1 + we) - 1) %R →
((1 + xf) × (1 + yf) × (1 + wf) - 1) ≤ zf %R →
Ein.rel_R we wf xr (xy + yr) %R →
Ein.rel_R xe xf (xx + yy).

Lemma Ein.absol_R:
∀ (xe xf ye yf xe xf ye yf : error) (xr yr xe : base) (xx yy : R),
Ein.absol_R xe xf xx xx → Ein.absol_R ye yf yr yy →
Ein.bound_R xl xu yr yu xx yy →
let (ze, zf) := Imult_R_fun (Imult_R_fun (Imult_R_fun (xe, xf) (yl, yu)) (Imult_R_fun (ye, yf) (xl, xu))) in
Ein.absol_R xe zf (xy + yr) %R (xx + yy) %R.

End Ebase2.

Module Efloat_r2ec

Add LoadPath "Float".
Require AllFloat.
Require Ebase.
Require ClosestCalc.
Require Ibase_R.
Require Float_bonus.

Section Efloat_r2ec.

Definition radix := 2**.

Local radix More Than One := Two_More Than One.
Lemma radix Not Zero : '(0 < radix).
Variable precision : nat.
Hypothesis precision More Than One: (lt (1 precision).
Lemma precision Not Zero: '(precision ≠ (0)).
Variable bExp : nat.
Definition bNum := (iter_nat precision positive x0 xH).
Definition bound := (Bound bNum bExp).
Lemma pGivesBound: (POS (uNum bound)) → (Zpower_nat '2 precision).
Record cFloat: Set := CFloat {
value: float;
cFloat_canonic: (Feanonic radix round value) }.

Coercion Local cFloat2R := [x:cFloat] (FtoR radix (value x)).
Coercion Local float2R := (FtoR radix).

Definition Eint f2 := (Eint cFloat cFloat2R float float2R).
Definition Eint f2_bound := (Eint bound cFloat cFloat2R).
Definition Eint f2_error := (Eint error cFloat cFloat2R float float2R).

Definition Eint f2_inclusion_helper := [x,y]:cFloat
(andb
(Fle Bool radix (value y) (value x))
(Fle Bool radix (value x) (value y))).

Lemma Eint f2_inclusion: (x,y):cFloat
(Eint f2_bound x y) →
(Eint f2_inclusion_helper x y) → true →
(Eint f2_bound x y).

Definition Eint f2_union_helper := [a,b]:cFloat
(Feu Bool radix (FSucc bound radix precision a) (value b)).

Lemma Eint f2_union: (P:Prop) (a,b,c):cFloat
(Eint f2_union_helper a b c) →
(Eint f2_bound a b c) →
(Eint f2_bound a b c) → P →
(Eint f2_bound a b c) → P → P.

Definition Rounded := [a,R]:cFloat (EvenClosest bound radix precision a (value b)).

Definition cFloat_plus := [a,b,c]:cFloat
(Rounded "a+b" c).

Definition cFloat_minus := [a,b,c]:cFloat
(Rounded "a-b" c).

Definition cFloat_mult := [a,b,c]:cFloat
(Rounded "a×b" c).

Definition cFloat_div := [a,b,c]:cFloat
(Rounded "a/b" c).

Definition cFloat_error :=
[a,b]:cFloat (F dot 1 (Zmax (Fexp (value a)) (Fexp (value b)))) - 1).

Definition RError := [x]:cFloat (F dot 1 (Fexp (value x)) - 1).

Lemma RError_correct: (x,y):cFloat
(Rounded y x) → "d(Rabsoluto x y) ≤ (RError x).".

Lemma RError_rng_correct_aux: (x,y):cFloat
"d(Rabsoluto x) ≤ (Rabsoluto y) → "d(RError x) ≤ (RError y).".

Lemma RError_rng_correct: (x,y,z):cFloat (e:Float)
"x ≤ y → "z ≤ y → "d(RError x) ≤ e → "d(RError y) ≤ e →
"d(RError x) ≤ e.".

Lemma Rounded_monotone:
((x,y):R) (a,b:Float) "x ≤ y → (Rounded x a) → (Rounded y b) → "a ≤ b".

Lemma cFloat_error RError_l: (x,y,z):cFloat "d(RError x) ≤ (cFloat_error x y).".

Lemma cFloat_error RError_r: (x,y,z):cFloat "d(RError y) ≤ (cFloat_error x y).".

Lemma cFloat_error RError: (x,y,z):cFloat
(Eint f2_bound x y z) → "d(RError x) ≤ (cFloat_error x y)."

Definition Eplus f2ec_bound_helper :=
(x,y,z):cFloat (evaluate_even_closest radix bound precision)
(Fplus radix (value xl) (value yl)) (value zl))
(evaluate_even_closest radix bound precision
 (Fplus radix (value xu) (value yu)) (value zl))).

Definition Eplus_f2ec_error_helper :=
[xe, ye, xe'float] [xu, xu'float]
(Fle_bool radix (Fplus radix (Fplus radix xe ye) (cFloat_error xl zu)) xe).

Lemma Eplus_f2ec_bound_aus:
(xe, xe yl, xu, xu'float) (x, y, x'cFloat)
(Eint_f2_bound xl xu x) → (Eint_f2_bound yl yu y) →
(cFloat_plus x x y x) → (cFloat_plus xl yl x) → (cFloat_plus zu yu zu) →
(Eint_f2_bound xl zu zu).

Lemma Eplus_f2ec_bound:
(xe, xe yl, xu, xu'float) (x, y, x'cFloat)
(Eint_f2_bound xl xu x) → (Eint_f2_bound yl yu y) →
(cFloat_plus x x y x) →
(Eplus_f2ec_bound_helper xl xu yl yl zl true) →
(Eint_f2_bound xl zu zu).

Lemma Eplus_f2ec_error:
(xe, xe yl, xu, xu'float) (x, y, x'cFloat) (x, y, y'cFloat) (x, y, y'R)
(Eint_f2_error x x x) → (Eint_f2_error y y y) →
(cFloat_plus x x y x) → (Eint_f2_bound zl zl x) →
(Eplus_f2ec_error_helper xe ye ze zl zu) →
(Eint_f2_error ze x z + y y).

Lemma Eminus_le_2: (a, b, c, d: R) "a ≤ b" → "c ≤ d" → "a - d ≤ b - c".

Definition Eminus_f2ec_bound_helper :=
[xe, xe yl, xu, xu'float]
(and
(evaluate_even_closest radix bound precision
 (Fminus radix (value xl) (value yl)) (value zl))
(evaluate_even_closest radix bound precision
 (Fminus radix (value xu) (value yl)) (value zu))).

Definition Eminus_f2ec_error_helper := Eplus_f2ec_error_helper.

Lemma Eminus_f2ec_bound_aus:
(xe, xe yl, xu, xu'float) (x, y, x'cFloat)
(Eint_f2_bound xl xu x) → (Eint_f2_bound yl yu y) →
(cFloat_minus x x y x) → (cFloat_minus xl yl x) → (cFloat_minus zu yu zu) →
(Eint_f2_bound xl zu zu).

Lemma Eminus_f2ec_bound:
(xe, xe yl, xu, xu'float) (x, y, x'cFloat)
(Eint_f2_bound xl xu x) → (Eint_f2_bound yl yu y) →
(cFloat_minus x x y x) → (Eminus_f2ec_bound_helper xl xu yl yl zl true) →
(Eint_f2_bound xl zu zu).

Lemma Eminus_f2ec_error:
(xe, xe yl, xu, xu'float) (x, y, x'cFloat) (x, y, x'cFloat) (x, y, y'R)
(Eint_f2_error x x x) → (Eint_f2_error y y y) →
(cFloat_minus x x y x) → (Eint_f2_bound zl zl x) →
(Eminus_f2ec_error_helper xe ye ze zl zu) →
(Eint_f2_error ze x z - y y).

Definition Eplus_exact_f2ec_bound_helper :=
[xe, xe yl, xu, xu'float]
(and
(Feq_bool radix (Fplus radix (value xl) (value yl)) (value zl))
(Eint_f2_error (Fplus radix xe ye) zr "xx - yy")

Lemma Eminus_exact_f2ec_error:
(xl,ze:Float;xe:float) (yl,ze:Float;ye:float) (xl,ze:Float;ze:float)
(xx,yy:R)
(Eint_f2 xl xu xe xx) → (Eint_f2 xl xu xe xx)
(c_Float_minus xx yr zr) → (Eint_f2_bound xl xu xe xx) →
(Eplus_exact_f2ec_error_helper xl xu xe xl xu xe) → true →
(Eint_f2_error xe zr "xx - yy")

Definition Emult_exact_f2ec_bound_helper :=
[xl,xu,yl,yu,zl,zu:Float]
let il := (Fmult (value xl) (value yl)) in
let lu := (Fmult (value xl) (value yu)) in
let ul := (Fmult (value xu) (value yl)) in
let uu := (Fmult (value xu) (value yu)) in
let l := (Fmin_radix (Fmin_radix il lu) (Fmin_radix ul uu)) in
let u := (Fmax_radix (Fmax_radix il lu) (Fmax_radix ul uu)) in

(Emult_exact_f2ec_bound_helper xl xu yl yu zl zu) → true →
(Eint_f2_bound il xu xe xx).

Local min_mult := [xl,xu,yl,yu:Float] (Fmin_radix
(Fmin_radix (Fmult (value xl) (value yl)) (Fmult (value xl) (value yu)))
(Fmin_radix (Fmult (value xu) (value yl)) (Fmult (value xu) (value yu))))

Local max_mult := [xl,xu,yl,yu:Float] (Fmax_radix
(Fmax_radix (Fmult (value xl) (value yl)) (Fmult (value xl) (value yu)))
(Fmax_radix (Fmult (value xu) (value yl)) (Fmult (value xu) (value yu))))

Lemma Emult_exact_f2ec_bound:
(xl,xu,yl,yu,zl,zu:Float) (xl,xu,yl,yu:Float)
(Eint_f2_bound xl xu xe xx) → (Emult_exact_f2ec_bound il xu ye yu)
(c_Float_mult xx yr zr) →
(Emult_exact_f2ec_bound_helper xl xu ye yu zl zu) → true →
(Eint_f2_bound il xu xe xx).

Definition Emult_exact_f2ec_error_helper :=
[xe,ye,xe:float] [xe,ye:float] [zl,zu:Float]
(Fxpool_radix (Fplus radix (Fmult xe ye) (Fmult xe ye))
(Fmult xe ye))
(c_Float_error zl zu) xe).

Lemma Emult_exact_f2ec_error_aux:
(xe,ye,xe:float) (xe,xe:float) (xl,xu,yl,yu:Float) (xl,xu,yl,yu:Float)
(Eint_f2_error xe xe xx) → (Eint_f2_error ye ye yy) →
(c_Float_mult xx yr zr) → (Eint_f2_bound xx xe xx)
\quad "(Rabsoluxr) ≤ zn → "(Rabsolu yr) ≤ yn →
(Emult_exact_f2ec_error_helper xe ye xe ye (value zn) (value yu) zl zu) → true →
(Eint_f2_error xe xe xx yx).

Lemma Emult_exact_f2ec_error:
(xl,xu,yl,yu:Float) (xl,xu,yl,yu:Float) (xl,xu,yl,yu:Float)
(xx,yy:R)
(Eint_f2 xl xu xe xx) → (Eint_f2 xl xu xe xx)
(c_Float_mult xx yr zr) → (Eint_f2_bound xx xu xe xx)
(Emult_exact_f2ec_error_helper xe ye ye xe ye (Fmax_radix (value xl) (value yu))
(Fmax_radix (value xl) (value yu) zl zu) → true →
(Eint_f2_bound xe xe xx yy).

Axiom Emult_exact_f2ec_bound:
(xl,xu,yl,yu,sl,zu::Float) (xr,yr,xr::Float)
(Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
(c Float\_div xr yr xr) →
true → true →
(Eint_f2_bound xl xu xr).

Axiom Edinv\_f2ec\_error:
(xl,xu,yl,yu,sl,zu::Float) (yl,yu::Float; ye::Float) (d,zu::Float; ze::Float)
(xr,yr,rx::Float) (xx,yy::R)
(Eint_f2 xl xu xx xr xx) → (Eint_f2 yl yu ye yr yy) →
(c Float\_div xr yr xr) → (Eint_f2_bound xl xu xr) →
true → true →
(Eint_f2\_error ze xr "xx / yy").

End Efloat\_r2ec.

Module Efloat\_s754ec

Add LoadPath "Float".
Require Reals.
Require AllFloat.
Require Efloat\_r2ec.
Require ClosestCalc.
Require Export Float.

Section Efloat\_s754ec.

Local precision := 'N:24'.
Local precision MoreThanOne: (lt (1) precision).
Unfold precision.
Auto with zarith.
Qed.
Local b Exp := 'N:149'.
Definition Eint\_s754 := (Eint\_f2 precision b Exp).
Definition Eint\_s754\_bound := (Eint\_f2\_bound precision b Exp).
Definition Eint\_s754\_error := (Eint\_f2\_error precision b Exp).
Definition Eint\_s754\_union := (Eint\_f2\_union precision precision MoreThanOne b Exp).
Definition Eint\_s754\_inclusion := (Eint\_f2\_inclusion precision b Exp).

Definition Eplus\_s754\_error := (Eplus\_f2ec\_error precision precision MoreThanOne b Exp).
Definition Eplus\_s754\_bound := (Eplus\_f2ec\_bound precision precision MoreThanOne b Exp).
Definition Eminus\_s754\_error := (Eminus\_f2ec\_error precision precision MoreThanOne b Exp).
Definition Eminus\_s754\_bound := (Eminus\_f2ec\_bound precision precision MoreThanOne b Exp).
Definition Emul\_s754\_error := (Emul\_f2ec\_error precision MoreThanOne b Exp).
Definition Emul\_s754\_bound := (Emul\_f2ec\_bound precision MoreThanOne b Exp).

Definition Ediv\_s754\_error := (Ediv\_f2ec\_error precision b Exp).
Definition Ediv\_s754\_bound := (Ediv\_f2ec\_bound precision b Exp).

Definition Eplus\_exact\_s754\_error := (Eplus\_exact\_f2ec\_error precision precision MoreThanOne b Exp).
Definition Eplus\_exact\_s754\_bound := (Eplus\_exact\_f2ec\_bound precision precision MoreThanOne b Exp).
Definition Eminus\_exact\_s754\_error := (Eminus\_exact\_f2ec\_error precision precision MoreThanOne b Exp).
Definition Eminus\_exact\_s754\_bound := (Eminus\_exact\_f2ec\_bound precision precision MoreThanOne b Exp).

Definition float\_s754 := (c Float precision b Exp).
Definition Good\_s754 := (CEqual (precision b Exp)).
Definition Float\_s754 := [f::float p:(Good\_s754 f)] (CFloat precision b Exp f p).
Definition Plus\_s754ec := (c Float\_plus precision b Exp).
Definition Minus\_s754ec := (c Float\_minus precision b Exp).
Definition Mul\_s754ec := (c Float\_mult precision b Exp).
Definition Div\_s754ec := (c Float\_div precision b Exp).

End Efloat\_s754ec.
Module IbaseR

Require Import Reals.

Section IbaseR.

Definition Int (xl xu x : R) := (xl ≤ x ≤ xu)%R.

Lemma IplusR :
∀ xl xu yl yu xl x y : R,
(xl ≤ xl + yl)%R → (xu + yu ≤ zu)%R →
Int xl xu x → Int yl yu y →
Int xl xu (x + y).

Definition IplusR_fun (xi yi : R×R) :=
let (xl,xu) := xi in let (yl,yu) := yi in
((xl + yl)%R, (xu + yu)%R).

Lemma IplusR_fun_correct :
∀ xl xu yl yu x y : R,
Int xl xu x → Int yl yu y →
let (zl,zu) := IplusR_fun (xl,xu) (yl,yu) in
Int zl zu (x + y).
intros xl xu yl yu x y lx ly.
simpl.
apply IplusR with xl xu yl yu; auto with real.
Qed.

Lemma IoppR_exact :
∀ xl xu x : R, Int xl xu x → Int (- xu) (- xl) (- x).

Lemma IoppR_exact_tuv :
∀ xl xu x : R, Int (- xu) (- xl) (- x) → Int xl xu x.

Lemma Rle_trans_and : ∀ a b c : R, (a ≤ b ≤ c)%R → (a ≤ c)%R.

Lemma monotony_1p :
∀ a x y : R, (0 ≤ a)%R → (x ≤ y)%R → (a × x ≤ a × y)%R.

Lemma monotony_2p :
∀ a x y : R, (0 ≤ a)%R → (x ≤ y)%R → (x × a ≤ y × a)%R.

Lemma monotony_1n :
∀ a x y : R, (a ≤ 0)%R → (x ≤ y)%R → (a × y ≤ a × x)%R.

Lemma monotony_2n :
∀ a x y : R, (a ≤ 0)%R → (x ≤ y)%R → (y × a ≤ x × a)%R.

Lemma ImulR_pp :
∀ xl yl yu xl zu x y : R,
(0 ≤ xl)%R → (0 ≤ yl)%R →
(xl ≤ xl × yl)%R → (zu × yu ≤ zu)%R →
Int xl zu x → Int yl yu y →
Int xl zu (x × y).

Lemma ImulR_pem :
∀ xl yl yu xl zu x y : R,
(0 ≤ xl)%R → (yl ≤ 0)%R → (0 ≤ yu)%R →
(xl ≤ xl × yl)%R → (zu × yu ≤ zu)%R →
Int xl zu x → Int yl yu y →
Int xl zu (x × y).

Lemma ImulR_pn :
∀ xl yl yu xl zu x y : R,
(0 ≤ xl)%R → (yu ≤ 0)%R →
\[(zl \leq xu \times yu)\% R \rightarrow (zl \times yu \leq zu)\% R \rightarrow \int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y)\].

**Lemma ImulIR\_np**:
\[
\forall zl xu yu zl zu x y : R,
(zl \leq 0)\% R \rightarrow (0 \leq xu)\% R \rightarrow (yu \leq 0)\% R \rightarrow (0 \leq yu)\% R \rightarrow
(zl \leq xu \times yu)\% R \rightarrow (zl \times yu \leq zu)\% R \rightarrow
(zl \times yu \leq zu)\% R \rightarrow (zl \times yu \leq zu)\% R \rightarrow
\int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y).
\]

**Lemma ImulIR\_mn**:
\[
\forall zl xu yu zl zu x y : R,
(zl \leq 0)\% R \rightarrow (0 \leq xu)\% R \rightarrow (yu \leq 0)\% R \rightarrow
(zl \leq xu \times yu)\% R \rightarrow (zl \times yu \leq zu)\% R \rightarrow
\int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y).
\]

**Lemma ImulIR\_np**:
\[
\forall zl xu yu zl zu x y : R,
(zl \leq 0)\% R \rightarrow (0 \leq xu)\% R \rightarrow (yu \leq 0)\% R \rightarrow
(zl \leq xu \times yu)\% R \rightarrow (zl \times yu \leq zu)\% R \rightarrow
\int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y).
\]

**Lemma ImulIR\_mn**:
\[
\forall zl xu yu zl zu x y : R,
(zu \leq 0)\% R \rightarrow (yl \leq 0)\% R \rightarrow (zl \leq zl \times yl)\% R \rightarrow
\int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y).
\]

**Lemma ImulIR\_mn**:
\[
\forall zl xu yu zl zu x y : R,
(zu \leq 0)\% R \rightarrow (yl \leq 0)\% R \rightarrow (zl \leq zl \times yl)\% R \rightarrow
\int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y).
\]

**Lemma ImulIR\_mn**:
\[
\forall zl xu yu zl zu x y : R,
(zu \leq 0)\% R \rightarrow (yl \leq 0)\% R \rightarrow (zl \leq zl \times yl)\% R \rightarrow
\int xu x \rightarrow \int yu y \rightarrow \int zl zu (x \times y).
\]

**Inductive interval\_sign (l u : R) : Set :=**
\[
| Nsign : (u \leq 0)\% R \rightarrow interval\_sign l u
| Msign : (l \leq 0)\% R \rightarrow (0 \leq u)\% R \rightarrow interval\_sign l u
| Psign : (0 \leq l)\% R \rightarrow interval\_sign l u.
\]

**Lemma interval\_sign\_correct** : \(\forall l u : R, (l \leq u)\% R \rightarrow interval\_sign l u\).

**Lemma Int\_sign** : \(\forall l u x : R, \int l u x \rightarrow interval\_sign l u\).

**Lemma Rmin\_trans** :
\[
\forall a b c : R, (c \leq Rmin a b)\% R \rightarrow (c \leq a)\% R \land (c \leq b)\% R.
\]

**Lemma Rmax\_trans** :
\[
\forall a b c : R, (Rmax a b \leq c)\% R \rightarrow (a \leq c)\% R \land (b \leq c)\% R.
\]

**Lemma ImulIR** :
∀ z l xu yl yu zl zu x y : R,
( zl ≤ Rmin ( Rmin ( zl × yl) ( zl × yu)) ( Rmin ( xu × yl) ( xu × yu)))%R →
( Rmax ( Rmax ( zl × yl) ( zl × yu)) ( Rmax ( xu × yl) ( xu × yu)))%R →
Intl zl xu x → Intl yl yu y → Intl zl zu (x × y).

Definition ImultRFun ( zi yi : R×R) :=
let (zl,zu) := zi in let (yl,yu) := yi in
(( Rmin ( Rmin ( zl × yl) ( zl × yu)) ( Rmin ( xu × yl) ( xu × yu)))%R,
( Rmax ( Rmax ( zl × yl) ( zl × yu)) ( Rmax ( xu × yl) ( xu × yu)))%R).

Lemma ImultRFun_correct :
∀ zl xu yl yu x y : R,
Intl zl xu x → Intl yl yu y →
let (zl,zu) := ImultRFun (zl,zu) (yl,yu) in
Intl zl zu (x × y).
intros zl xu yl yu x y lx ly.
simpl.
apply ImultR with zl xu yl yu; auto with real.
Qed.

Lemma Rle_Rmult : ∀ x y : R, (0 < x)%R → (x ≤ y)%R → (/ y ≤ / x)%R.

Lemma InvR_0p :
∀ yl yu zl zu y : R,
(0 < yl)%R →
Intl yl yu y → (zl ≤ / yu)%R → (/ yl ≤ zu)%R → Intl zl zu (/ y).

Lemma InvR_0n :
∀ yl yu zl zu y : R,
(ylu < 0)%R →
Intl yl yu y → (zl ≤ / yu)%R → (/ yl ≤ zu)%R → Intl zl zu (/ y).

Lemma InvR :
∀ yl yu zl zu y : R,
(yu < 0)%R ∨ (0 < yl)%R →
Intl yl yu y → (zl ≤ / yu)%R → (/ yl ≤ zu)%R → Intl zl zu (/ y).

Lemma IdivR :
∀ zl xu yl yu zl zu x y : R,
(yu < 0)%R ∨ (yl > 0)%R →
(zl ≤ Rmin ( Rmin ( yl / yu) ( xlu / yu)) ( Rmin ( xu / yl) ( xu / yu)))%R →
( Rmax ( Rmax ( xlu / yu) ( xlu / yl)) ( Rmax ( xu / ylu) ( xu / ylu)))%R →
Intl zl xu x → Intl yl yu y →
Intl zl zu (x / y).

End IbaseR.