HAL
open science

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-02102116v1

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Generating formally certified bounds on values and round-off errors

Marc Daumas and
Guillaume Melquiond

Juillet 2004

Research Report N° 2004-36

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

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 hand-written pen-and-paper proofs contain errors and there is no doubt that some highly regarded hand-written 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 $\tilde{x} : \mathbb{F}$, with $\mathbb{F}$ being the set of hardware-representable numbers. We do not mean that $\tilde{x}$ is the rounded value of x, we mean that $\tilde{x}$ 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 $\tilde{f}(\tilde{x})$ that takes into account the round-off errors caused by the computation. Discretization and truncation errors have to be handled separately. We use $\tilde{x}$ 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 $\tilde{f}(\tilde{x})$, the difference between the exact value $f(x)$ and the computed value $\tilde{f}(\tilde{x})$, and so on, when some properties are known on the input variables. For example, under the hypothesis $\tilde{x} \in [1, 4]$, it is easy to prove that $\sqrt{\tilde{x}}$ 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 independant 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. <= x)`, `x` refers to the value actually in memory or in a register; consequently the assertion does not apply to x but to $\tilde{x}$. When a variable must be proved not to underflow or overflow, we mean to check the property on $\tilde{x}$, and so on. Most of the properties proved at this stage require knowing the domain of $\tilde{x}$; 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 $\tilde{x}$ are in $\mathbb{F}$ as the approximated value $\tilde{x}$ 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 $\tilde{x}$ 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:

$$[\tilde{a}, \tilde{b}] + [\tilde{c}, \tilde{d}] = [\bigtriangledown(\tilde{a} + \tilde{c}), \bigtriangleup(\tilde{b} + \tilde{d})].$$

There $\bigtriangledown$ and $\bigtriangleup$ denote that operations should be rounded toward $-\infty$ and $+\infty$. The bounds may not carry enough precision for $\tilde{a} + \tilde{c}$ and $\tilde{b} + \tilde{d}$ 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 `Eint_bound` in the Coq support library.

In our cases, such rounding considerations are not necessary: the hardware-implemented addition $\boxplus$ 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

$$\tilde{x} \in [\tilde{a}, \tilde{b}] \quad \text{and} \quad \tilde{y} \in [\tilde{c}, \tilde{d}] \quad \text{then} \quad x \boxplus y \in [\tilde{a} \boxplus \tilde{c}, \tilde{b} \boxplus \tilde{d}]$$

Indeed, the values $\tilde{a} \boxplus \tilde{c}$ and $\tilde{b} \boxplus \tilde{d}$ already lie in $\mathbb{F}$, there is no need for an additional rounding. Consequently, since the addition $\boxplus$ is a monotonous function of $\mathbb{F}^2 \to \mathbb{F}$ whatever the rounding mode, then $\tilde{a} \boxplus \tilde{c}$ and $\tilde{b} \boxplus \tilde{d}$ are the bounds.

$$[\tilde{a}, \tilde{b}] \boxplus [\tilde{c}, \tilde{d}] = [\tilde{a} \boxplus \tilde{c}, \tilde{b} \boxplus \tilde{d}].$$

There is no need to take instead possibly different and non-optimal values $\bigtriangledown(\tilde{a}+\tilde{c})$ and $\bigtriangleup(\tilde{b}+\tilde{d})$.[2]

Consequently, the most commonly used interval arithmetic libraries were not appropriate for our project. We chose the Boost parameterizable interval arithmetic library [5]. 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 $[1, \circ(\sqrt{5})]$. Standard interval arithmetic will then answer that the whole expression is in $[0, 2] \div [1, \circ(\sqrt{5})] = [0, 2]$. 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.

[2] This result is given by `Eplus_f2ec_bound` for generic floating point numbers.

Absolute error[3] η_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. η_0 is the rounding error between the exact result of $\tilde{x} + \tilde{y}$ and its computed value $\tilde{x} \boxplus \tilde{y}$.

$$\begin{aligned} z - \tilde{z} &= (x + y) - (\tilde{x} \boxplus \tilde{y}) \\ &= (x + y) - (\tilde{x} + \tilde{y} - \eta_0) \\ &= (x - \tilde{x}) + (y - \tilde{y}) + \eta_0 \\ &= \eta_x + \eta_y + \eta_0. \end{aligned}$$

We extend the definition to interval bounds. Intervals A_x and A_y are attached to variables x and y and respectively enclose η_x and η_y. Interval A_0 encloses all possible values for η_0.[4] 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].

$$\eta_z \in A_z \subseteq A_x + A_y + A_0$$

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,

$$\begin{array}{ccccccccc} -\frac{u}{2} & \leq & -\frac{u}{2+u} & \leq & \epsilon_1 & \leq & \frac{u}{2+u} & \leq & \frac{u}{2} \quad \text{and} \\ -\frac{u}{2} & \leq & -\frac{u}{2(1+u)} & \leq & \epsilon_2 & \leq & \frac{u}{2} & \leq & \frac{u}{2}. \end{array}$$

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,

$$\begin{array}{ccccccccc} -\frac{u}{2} & \leq & -\frac{u}{2+u} & \leq & \epsilon_1 & \leq & \frac{u}{2-u} & \not\leq & \frac{u}{2} \quad \text{and} \\ -\frac{u}{2} & \leq & -\frac{u}{2} & \leq & \epsilon_2 & \leq & \frac{u}{2} & \leq & \frac{u}{2}. \end{array}$$

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] This definition is given in Coq by `Eint_absol`, and by `Eint_error` for symmetric interval.

[4] This result is given in the general case by `Eplus_error` and for floating point numbers by `Eplus_f2ec_error` and `Eplus_exact_f2ec_error` depending on whether Sterbenz's conditions on exponents are verified.

[5] This definition is given by `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[6] is:

$$\begin{aligned} \frac{z}{\tilde{z}} &= \frac{x \times y}{\tilde{x} \boxtimes \tilde{y}} \\ &= \frac{x \times y}{\tilde{x} \times \tilde{y}}(1+\epsilon_0) \\ &= (1+\epsilon_x)(1+\epsilon_y)(1+\epsilon_0). \end{aligned}$$

Once again, we attach intervals R_x and R_y to respective variables x and y enclosing ϵ_x and ϵ_y. Let interval R_0 enclose the value of the relative error ϵ_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.

2.2.3 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 $A \div I$ 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.

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

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.

2.3 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:

$$\tilde{a} \boxplus \tilde{c} \leq \tilde{b} \boxplus \tilde{d} \quad \text{if} \quad \tilde{a} \leq \tilde{b} \quad \text{and} \quad \tilde{c} \leq \tilde{d}.$$

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 ProofLite[7] 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

[6] This result is given by `Emult_relat` in the general case.
[7] `http://research.nianet.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 $\tilde{a} \leq \tilde{b} \boxplus \tilde{c}$, the tools acts as an oracle and computes beforehand the sum $\tilde{d} = \tilde{b} \boxplus \tilde{c}$ and provides this number $\tilde{d}$ to the proof checker. Consequently, the checker does not have to explicitly compute $\tilde{b} \boxplus \tilde{c}$ in order to prove the property, it only has to verify that $\tilde{d}$ as provided by the tool is the closest[8] (with respect to the rounding mode of $\boxplus$) floating point number to $\tilde{b} + \tilde{c}$, and then that $\tilde{a} \leq \tilde{d}$. 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,

$$\tilde{c} = \circ(\tilde{a} \div \tilde{b}) \quad \text{iff} \quad \tilde{b} \times \tilde{c} \ \text{ is closer from } \ \tilde{a} \ \text{ than } \ \tilde{b} \times \tilde{c}^- \ \text{ and } \ \tilde{b} \times \tilde{c}^+$$

where $\tilde{c}^-$ and $\tilde{c}^+$ are the floating point numbers respectively just below and just above $\tilde{c}$.

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 $\lceil m \cdot 2^{-k} \rceil \times 2^{e+k}$ for a given k. The bound is larger but certifying it is faster as $\lceil m \cdot 2^{-k} \rceil$ 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 ownship is just a point. A conflict is then defined as the incursion of the ownship 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 `evaluate_even_closest` for rounding to nearest and tie-breaking to even.

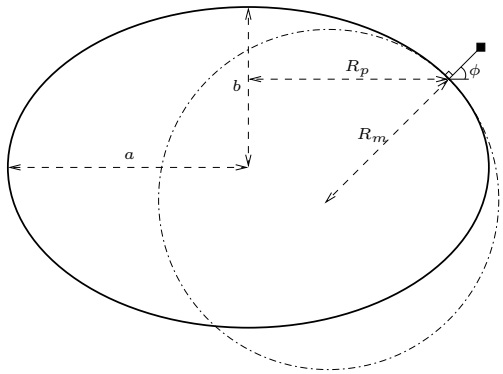

Figure 1: Geodetic latitude and radius in WGS84 system

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:

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

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: ϕ_1, ϕ_2, λ_1, λ_2, v_{N1}, v_{N2}, v_{E1}, v_{E2}

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

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/`.

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 $\hat{R}_p(\phi)$ approximating $R_p(\phi)$

Input: ϕ {all the dyadic numbers are floating point numbers}

$$
\begin{aligned}
x &\leftarrow 511225 \times 2^{-18} - \phi^2 \\
y &\leftarrow 4439091 \cdot 2^{-2} + x \times (9023647 \cdot 2^{-2} + x \times (\\
&\quad 13868737 \cdot 2^{-6} + x \times (13233647 \cdot 2^{-11} + x \times (\\
&\quad -1898597 \cdot 2^{-14} + x \times (-6661427 \cdot 2^{-17}))))
\end{aligned}
$$

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_x and s_y is bounded by half a meter, and the error on v_{x1} and v_{x2} is bounded by 5×10^{-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 `Efloat_s754ec`.

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

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[2] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[3] Sylvie Boldo and Marc Daumas. Properties of two's complement floating point notations. *International Journal on Software Tools for Technology Transfer*, 5(2-3):237–246, 2004.

[4] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 2004.

[5] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The Boost interval arithmetic library. In *Real Numbers and Computers*, pages 65–80, Lyon, France, 2003.

[6] Marc Daumas, Laurence Rideau, and Laurent 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.

[7] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[8] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, Université Paris Sud, 2003.

[9] Debbie Gage and John McCormick. We did nothing wrong. *Baseline*, 1(28):32–58, 2004.

[10] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

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

[12] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 1996.

[13] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant: a tutorial: version 7.2. Technical Report 256, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France, 2002.

[14] Information Management and Technology Division. Patriot missile defense: software problem led to system failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992.

[15] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied interval analysis*. Springer, 2001.

[16] William Kahan. Lecture notes on the status of the IEEE standard 754 for binary floating point arithmetic. Published on the net, 1996.

[17] Leslie Lamport and P. Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[18] Ren-Cang Li, Sylvie Boldo, and Marc Daumas. Theorems on efficient argument reductions. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 129–136, Santiago de Compostela, Spain, 2003.

[19] Jacques-Louis Lions et al. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996.

[20] Paul S. Miner. *Hardware verification using coinductive assertions.* PhD thesis, Indiana University, Bloomington, Indiana, 1998.

[21] Chris Moody and Gary Furr. Automatic dependent surveillance broadcast. Section 2.2, Draft 12 UAT-WP-12-07, Radio Technical Commission for Aeronautics, 2002.

[22] César Muñoz, Víctor Carreño, Gilles Dowek, and Ricky Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3):371–380, 2003.

[23] Arnold Neumaier. *Interval methods for systems of equations.* Cambridge University Press, 1990.

[24] Isaac Newton. *La méthode des fluxions et des suites infinies.* de Bure l'aîné, 1740. Traduit par de Buffon.

[25] Sylvie Putot, Eric Goubault, and Matthieu Martel. Static analysis based validation of floating point computations. In *Novel Approaches to Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 306–313, Dagstuhl, Germany, 2004.

[26] Joseph Raphson. *Analysis aequationum universalis.* 1690.

[27] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *Proceedings of the Conference on Software for Critical Systems*, pages 1–15, New Orleans, Louisiana, 1991.

[28] Pat H. Sterbenz. *Floating point computation.* Prentice Hall, 1974.

[29] Ping Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, 1991.

Module ClosestCalc

```
Add LoadPath "Float".
Require AllFloat.

Section ClosestCalc.
Variable radix: Z.
Variable bound: Fbound.
Variable precision: nat.

Coercion Local FtoRradix := (FtoR radix).
Hypothesis radixMoreThanOne : (Zlt (POS xH) radix).

Local radixNotZero := (Zlt_1_O ? (Zlt_le_weak ? ? radixMoreThanOne)).
Hypothesis precisionNotZero : ¬ precision = O.
Hypothesis precisionMoreThanOne: (lt (1) precision).
Hypothesis pGivesBound : (POS (vNum bound)) = (Zpower_nat radix 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 precision 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 radix dd (dp x r f)) of
 | INFERIEUR ⇒ bad
 | EGAL ⇒    Cases (Fcompare radix dd (ds x r f)) of
  | INFERIEUR ⇒ bad
  | EGAL ⇒ tie
  | SUPERIEUR ⇒ tie
  end
 | SUPERIEUR ⇒
  Cases (Fcompare radix dd (ds x r f)) of
  | INFERIEUR ⇒ bad
  | EGAL ⇒ tie
  | SUPERIEUR ⇒ good
  end
 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)
 (Fcanonic radix bound x) → (is_float_even x) = true → (FNeven bound radix precision x).

Definition evaluate_even_closest := [x,r:float]
 Cases (evaluate_rounding 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 → (Fcanonic 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_coerc: base → R.

Coercion Local base2R := base_coerc.

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

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

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

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

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

Variable error: Set.
Variable error_coerc: error → R.
Coercion Local error2R := error_coerc.

Definition base_plus_e := [a,b,c,d:base] [e:error] (x,y,z:base)
 "a ≤ x" → "x ≤ c" → "b ≤ y" → "y ≤ d" →
 (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)
```

“$a \leq x$“ → “$x \leq c$“ → “$b \leq y$“ → “$y \leq d$“ →
($base_minus\ x\ y\ z$) → “($Rabsolu$ (x-y - z)) $\leq e$“.

Definition $base_mult_e$:= [a,b,c,d:$base$] [e:$error$] (x,y,z:$base$)
“$a \leq x$“ → “$x \leq c$“ → “$b \leq y$“ → “$y \leq d$“ →
($base_mult\ x\ y\ z$) → “($Rabsolu$ ($x \times y$ - z)) $\leq e$“.

Definition $Eint_bound$:= [xl,xu:$base$] [xr:$base$] “$xl \leq xr \leq xu$“.
Definition $Eint_error$:= [xe:$error$] [xr:$base$] [xx:R] “($Rabsolu$ (xx - xr)) $\leq xe$“.

Lemma $Eint_error_trans$:
(me,xe:$error$;xr:$base$;xx:R)
($Eint_error\ me\ xr\ xx$) → ($Rle\ me\ xe$) → ($Eint_error\ xe\ xr\ xx$).

Definition $Eint$:= [xl,xu:$base$] [xe:$error$] [xr:$base$] [xx:R]
($Eint_bound\ xl\ xu\ xr$) ∧ ($Eint_error\ xe\ xr\ xx$).

Lemma $Eplus_error_aux$:
(xe,ye:$error$) (xr,yr:$base$) (xx,yy:R)
($Eint_error\ xe\ xr\ xx$) → ($Eint_error\ ye\ yr\ yy$) →
“($Rabsolu$ (($xx+yy$) - ($xr+yr$))) $\leq xe+ye$“.

Lemma $Eplus_error$:
($xe,ye,ze,ze1$:$error$) (xr,yr,zr:$base$) (xx,yy:R)
($Eint_error\ xe\ xr\ xx$) → ($Eint_error\ ye\ yr\ yy$) →
($base_plus\ xr\ yr\ zr$) →
“$xe + ye + ze1 \leq ze$“ → “($Rabsolu$ ($xr+yr$ - zr)) $\leq ze1$“ →
($Eint_error\ ze\ zr$ “$xx+yy$“).

Lemma $Eminus_error_aux$:
(xe,ye:$error$) (xr,yr:$base$) (xx,yy:R)
($Eint_error\ xe\ xr\ xx$) → ($Eint_error\ ye\ yr\ yy$) →
“($Rabsolu$ ((xx-yy) - (xr-yr))) $\leq xe+ye$“.

Lemma $Eminus_error$:
($xe,ye,ze,ze1$:$error$) (xr,yr,zr:$base$) (xx,yy:R)
($Eint_error\ xe\ xr\ xx$) → ($Eint_error\ ye\ yr\ yy$) →
($base_minus\ xr\ yr\ zr$) →
“$xe + ye + ze1 \leq ze$“ → “($Rabsolu$ (xr-yr - zr)) $\leq ze1$“ →
($Eint_error\ ze\ zr$ “xx-yy“).

Lemma $Emult_error_aux$:
(xe,ye:$error$) (xr,yr,xn,yn:$base$) (xx,yy:R)
($Eint_error\ xe\ xr\ xx$) → ($Eint_error\ ye\ yr\ yy$) →
“($Rabsolu\ xr$) $\leq xn$“ → “($Rabsolu\ yr$) $\leq yn$“ →
“($Rabsolu$ ($xx \times yy$ - $xr \times yr$)) $\leq xe \times yn + ye \times xn + xe \times ye$“.

Lemma $Emult_error$:
($xe,ye,ze,ze1$:$error$) (xr,yr,zr,xn,yn:$base$) (xx,yy:R)
($Eint_error\ xe\ xr\ xx$) → ($Eint_error\ ye\ yr\ yy$) →
($base_mult\ xr\ yr\ zr$) → “($Rabsolu$ ($xr \times yr$ - zr)) $\leq ze1$“ →
“($Rabsolu\ xr$) $\leq xn$“ → “($Rabsolu\ yr$) $\leq yn$“ →
“$xe \times yn + ye \times xn + xe \times ye + ze1 \leq ze$“ →
($Eint_error\ ze\ zr$ “$xx \times yy$“).

End $Ebase$.

Module Ebase2

Require Import $Reals$.
Require Import $IbaseR$.

Section $Ebase2$.

Variable $base$: Set.
Variable $base_plus$: $base \to base \to base \to Prop$.
Variable $base_minus$: $base \to base \to base \to Prop$.
Variable $base_mult$: $base \to base \to base \to Prop$.
Variable $base_coerc$: $base \to R$.

$Coercion$ Local $base2R := base_coerc$.

Definition $base_plus_l\ (a\ b\ c : base) :=$
$\forall\ x\ y\ z : base,$
$(a \leq x)\%R \to (b \leq y)\%R \to base_plus\ x\ y\ z \to (c \leq z)\%R.$

Definition $base_plus_u\ (a\ b\ c : base) :=$
$\forall\ x\ y\ z : base,$
$(x \leq a)\%R \to (y \leq b)\%R \to base_plus\ x\ y\ z \to (z \leq c)\%R.$

Definition $base_minus_l\ (a\ b\ c : base) :=$
$\forall\ x\ y\ z : base,$
$(a \leq x)\%R \to (y \leq b)\%R \to base_minus\ x\ y\ z \to (c \leq z)\%R.$

Definition $base_minus_u\ (a\ b\ c : base) :=$
$\forall\ x\ y\ z : base,$
$(x \leq a)\%R \to (b \leq y)\%R \to base_minus\ x\ y\ z \to (z \leq c)\%R.$

Definition $base_mult_lu\ (a\ b\ c\ d\ e\ f : base) :=$
$\forall\ x\ y\ z : base,$
$(a \leq x)\%R \to$
$(x \leq c)\%R \to$
$(b \leq y)\%R \to (y \leq d)\%R \to base_mult\ x\ y\ z \to (e \leq z \leq f)\%R.$

Variable $error$: Set.
Variable $error_coerc$: $error \to R$.
$Coercion$ Local $error2R := error_coerc$.

Definition $base_plus_e\ (a\ b\ c\ d : base)\ (e : error) :=$
$\forall\ x\ y\ z : base,$
$(a \leq x)\%R \to$
$(x \leq c)\%R \to$
$(b \leq y)\%R \to (y \leq d)\%R \to base_plus\ x\ y\ z \to (Rabs\ (x + y - z) \leq e)\%R.$

Definition $base_minus_e\ (a\ b\ c\ d : base)\ (e : error) :=$
$\forall\ x\ y\ z : base,$
$(a \leq x)\%R \to$
$(x \leq c)\%R \to$
$(b \leq y)\%R \to (y \leq d)\%R \to base_minus\ x\ y\ z \to (Rabs\ (x - y - z) \leq e)\%R.$

Definition $base_mult_e\ (a\ b\ c\ d : base)\ (e : error) :=$
$\forall\ x\ y\ z : base,$
$(a \leq x)\%R \to$
$(x \leq c)\%R \to$
$(b \leq y)\%R \to (y \leq d)\%R \to base_mult\ x\ y\ z \to (Rabs\ (x \times y - z) \leq e)\%R.$

Definition $Eint_bound_R\ (xl\ xu\ xr : R) := (xl \leq xr \leq xu)\%R.$
Definition $Eint_absol_R\ (xe\ xf\ xr\ xx : R) :=$
$(xe \leq xx - xr \leq xf)\%R.$
Definition $Eint_relat_R\ (xe\ xf\ xr\ xx : R) :=$
$xr \neq 0\%R :> R \wedge (0 < 1 + xe)\%R \wedge (1 + xe \leq xx\ /\ xr \leq 1 + xf)\%R.$
Definition $Eint_bound\ (xl\ xu\ xr : base) := (Eint_bound_R\ xl\ xu\ xr).$
Definition $Eint_absol\ (xe\ xf : error)\ (xr : base)\ (xx : R) :=$
$(Eint_absol_R\ xe\ xf\ xr\ xx).$
Definition $Eint_relat\ (xe\ xf : error)\ (xr : base)\ (xx : R) :=$
$(Eint_relat_R\ xe\ xf\ xr\ xx).$

Lemma $Eplus_absol_R$:

∀ (xe xf ye yf xr yr xx yy : R),
Eint_absol_R xe xf xr xx →
Eint_absol_R ye yf yr yy →
Eint_absol_R (xe + ye)%R (xf + yf)%R (xr + yr)%R (xx + yy)%R.

Lemma Eplus_absol :
∀ (xe xf ye yf ze zf we wf : error) (xr yr zr : base) (xx yy : R),
Eint_absol xe xf xr xx → Eint_absol ye yf yr yy →
base_plus xr yr zr →
(ze ≤ xe + ye + we)%R → (xf + yf + wf ≤ zf)%R →
Eint_absol_R we wf zr (xr + yr)%R →
Eint_absol ze zf zr (xx + yy).

Lemma Emult_relat_R :
∀ (xe xf ye yf xr yr xx yy : R),
Eint_relat_R xe xf xr xx → Eint_relat_R ye yf yr yy →
Eint_relat_R ((1 + xe) × (1 + ye) - 1)%R ((1 + xf) × (1 + yf) - 1)%R (xr × yr)%R (xx × yy)%R.

Lemma Emult_relat :
∀ (xe xf ye yf ze zf we wf : error) (xr yr zr : base) (xx yy : R),
Eint_relat xe xf xr xx → Eint_relat ye yf yr yy →
base_mult xr yr zr → (base_coerc zr) ≠ 0%R → (0 < 1 + ze)%R →
(ze ≤ (1 + xe) × (1 + ye) × (1 + we) - 1)%R →
((1 + xf) × (1 + yf) × (1 + wf) - 1 ≤ zf)%R →
Eint_relat_R we wf zr (xr × yr)%R →
Eint_relat ze zf zr (xx × yy).

Lemma Emult_absol_R :
∀ (xe xf ye yf xr xl xu yr yl yu xx yy : R),
Eint_absol_R xe xf xr xx → Eint_absol_R ye yf yr yy →
Eint_bound_R xl xu xr → Eint_bound_R yl yu yr →
let (ze,zf) := IplusR_fun (IplusR_fun (ImultR_fun (xe,xf) (yl,yu)) (ImultR_fun (ye,yf) (xl,xu))) (ImultR_fun (xe,xf) (ye,yf)) in
Eint_absol_R ze zf (xr × yr)%R (xx × yy)%R.

End Ebase2.

Module Efloat_r2ec

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

Section Efloat_r2ec.

Definition radix := `2`.

Local radixMoreThanOne := TwoMoreThanOne.
Lemma radixNotZero: `0 < radix`.

Variable precision: nat.
Hypothesis precisionMoreThanOne: (lt (1) precision).
Lemma precisionNotZero: ~(precision = (0)).

Variable bExp: nat.
Definition bNum := (iter_nat precision positive xO xH).
Definition bound := (Bound bNum bExp).

Lemma pGivesBound: (POS (vNum bound)) = (Zpower_nat `2` precision).

Record cFloat: Set := CFloat {

```
  value: float;
  cFloat_canonic: (Fcanonic radix bound 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 := [xl,xu,yl,yu:cFloat]
 (andb
  (Fle_bool radix (value yl) (value xl))
  (Fle_bool radix (value xu) (value yu))).

Lemma Eint_f2_inclusion: (xl,xu,yl,yu,yr:cFloat)
 (Eint_f2_bound xl xu yr) →
 (Eint_f2_inclusion_helper xl xu yl yu) = true →
 (Eint_f2_bound yl yu yr).

Definition Eint_f2_union_helper := [a,b:cFloat]
 (Feq_bool radix (FSucc bound radix precision (value a)) (value b)).

Lemma Eint_f2_union: (P:Prop) (xl,xm,xn,xu,xr:cFloat)
 (Eint_f2_union_helper xm xn) = true → (Eint_f2_bound xl xu xr) →
 ((Eint_f2_bound xl xm xr) → P) →
 ((Eint_f2_bound xn xu xr) → P) → P.

Definition Rounded := [a:R;b: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] (Float '1' '(Zmax (Fexp (value a)) (Fexp (value b))) - 1').

Definition RError := [x:cFloat] (Float '1' '(Fexp (value x)) - 1').

Lemma RError_correct: (x:cFloat;y:R)
 (Rounded y x) →"(Rabsolu (y - x)) ≤ (RError x)".

Lemma RError_range_correct_aux: (x,y:cFloat)
 "(Rabsolu x) ≤ (Rabsolu y)" → "(RError x) ≤ (RError y)".

Lemma RError_range_correct: (x,y,z:cFloat) (e:float)
 "x ≤ z" → "z ≤ y" → "(RError x) ≤ e" → "(RError y) ≤ e" →
 "(RError z) ≤ e".

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

Lemma cFloat_error_RError_l: (x,y:cFloat) "(RError x) ≤ (cFloat_error x y)".

Lemma cFloat_error_RError_r: (x,y:cFloat) "(RError y) ≤ (cFloat_error x y)".

Lemma cFloat_error_RError: (x,y,z:cFloat)
 (Eint_f2_bound x y z) → "(RError z) ≤ (cFloat_error x y)".

Definition Eplus_f2ec_bound_helper :=
 [xl,xu,yl,yu,zl,zu:cFloat]
 (andb
  (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 zu))).
```

Definition *Eplus_f2ec_error_helper* :=
```
 [xe,ye,ze:float] [zl,zu:cFloat]
 (Fle_bool radix (Fplus radix (Fplus radix xe ye) (cFloat_error zl zu)) ze).
```

Lemma *Eplus_f2ec_bound_aux*:
```
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_plus xr yr zr) → (cFloat_plus xl yl zl) → (cFloat_plus xu yu zu) →
 (Eint_f2_bound zl zu zr).
```

Lemma *Eplus_f2ec_bound*:
```
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_plus xr yr zr) →
 (Eplus_f2ec_bound_helper xl xu yl yu zl zu) = true →
 (Eint_f2_bound zl zu zr).
```

Lemma *Eplus_f2ec_error*:
```
 (xe,ye,ze:float) (zl,zu:cFloat) (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2_error xe xr xx) → (Eint_f2_error ye yr yy) →
 (cFloat_plus xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Eplus_f2ec_error_helper xe ye ze zl zu) = true →
 (Eint_f2_error ze zr "xx + yy").
```

Lemma *Rminus_le_2*: (a,b,c,d:R) "a ≤ b" → "c ≤ d" → "a-d ≤ b-c".

Definition *Eminus_f2ec_bound_helper* :=
```
 [xl,xu,yl,yu,zl,zu:cFloat]
 (andb
  (evaluate_even_closest radix bound precision
   (Fminus radix (value xl) (value yu)) (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_aux*:
```
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_minus xr yr zr) → (cFloat_minus xl yu zl) → (cFloat_minus xu yl zu) →
 (Eint_f2_bound zl zu zr).
```

Lemma *Eminus_f2ec_bound*:
```
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_minus xr yr zr) →
 (Eminus_f2ec_bound_helper xl xu yl yu zl zu) = true →
 (Eint_f2_bound zl zu zr).
```

Lemma *Eminus_f2ec_error*:
```
 (xe,ye,ze:float) (zl,zu:cFloat) (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2_error xe xr xx) → (Eint_f2_error ye yr yy) →
 (cFloat_minus xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Eminus_f2ec_error_helper xe ye ze zl zu) = true →
 (Eint_f2_error ze zr "xx - yy").
```

Definition *Eplus_exact_f2ec_bound_helper* :=
```
 [xl,xu,yl,yu,zl,zu:cFloat]
 (andb
  (Feq_bool radix (Fplus radix (value xl) (value yl)) (value zl))
```

```
  (Feq_bool radix (Fplus radix (value xu) (value yu)) (value zu))).

Lemma Eplus_exact_f2ec_bound:
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_plus xr yr zr) →
 (Eplus_exact_f2ec_bound_helper xl xu yl yu zl zu) = true →
 (Eint_f2_bound zl zu zr).

Lemma Eplus_exact_f2ec_error_aux:
 (xe,ye:float) (zl,zu:cFloat) (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2_error xe xr xx) → (Eint_f2_error ye yr yy) →
 (cFloat_plus xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Zle (Fexp (value zl)) (Fexp (value xr))) →
 (Zle (Fexp (value zu)) (Fexp (value xr))) →
 (Zle (Fexp (value zl)) (Fexp (value yr))) →
 (Zle (Fexp (value zu)) (Fexp (value yr))) →
 (Eint_f2_error (Fplus radix xe ye) zr "xx + yy").

Definition Eplus_exact_f2ec_error_helper :=
 [xl,xu:cFloat;xe:float] [yl,yu:cFloat;ye:float] [zl,zu:cFloat;ze:float]
 let m = (Zmin (Fexp (Fmig radix bound (value xl) (value xu)))
               (Fexp (Fmig radix bound (value yl) (value yu)))) in
 (andb
  (Fle_bool radix (Fplus radix xe ye) ze)
 (andb
  (Zle_bool (Fexp (value zl)) m)
  (Zle_bool (Fexp (value zu)) m))).

Lemma Eplus_exact_f2ec_error:
 (xl,xu:cFloat;xe:float) (yl,yu:cFloat;ye:float) (zl,zu:cFloat;ze:float)
 (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2 xl xu xe xr xx) → (Eint_f2 yl yu ye yr yy) →
 (cFloat_plus xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Eplus_exact_f2ec_error_helper xl xu xe yl yu ye zl zu ze) = true →
 (Eint_f2_error ze zr "xx + yy").

Definition Eminus_exact_f2ec_bound_helper :=
 [xl,xu,yl,yu,zl,zu:cFloat]
 (andb
  (Feq_bool radix (Fminus radix (value xl) (value yu)) (value zl))
  (Feq_bool radix (Fminus radix (value xu) (value yl)) (value zu))).

Lemma Eminus_exact_f2ec_bound:
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_minus xr yr zr) →
 (Eminus_exact_f2ec_bound_helper xl xu yl yu zl zu) = true →
 (Eint_f2_bound zl zu zr).

Axiom cFloat_Fopp: (x:cFloat) (Fcanonic radix bound (Fopp (value x))).
Definition cFloat_opp := [x:cFloat]
 (CFloat (Fopp (value x)) (cFloat_Fopp x)).

Lemma Eminus_exact_f2ec_error_aux:
 (xe,ye:float) (zl,zu:cFloat) (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2_error xe xr xx) → (Eint_f2_error ye yr yy) →
 (cFloat_minus xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Zle (Fexp (value zl)) (Fexp (value xr))) →
 (Zle (Fexp (value zu)) (Fexp (value xr))) →
 (Zle (Fexp (value zl)) (Fexp (value yr))) →
 (Zle (Fexp (value zu)) (Fexp (value yr))) →
```

```
(Eint_f2_error (Fplus radix xe ye) zr "xx - yy").

Lemma Eminus_exact_f2ec_error:
 (xl,xu:cFloat;xe:float) (yl,yu:cFloat;ye:float) (zl,zu:cFloat;ze:float)
 (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2 xl xu xe xr xx) → (Eint_f2 yl yu ye yr yy) →
 (cFloat_minus xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Eplus_exact_f2ec_error_helper xl xu xe yl yu ye zl zu ze) = true →
 (Eint_f2_error ze zr "xx - yy").

Definition Emult_f2ec_bound_helper :=
 [xl,xu,yl,yu,zl,zu:cFloat]
 let ll = (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 ll lu) (Fmin radix ul uu)) in
 let u = (Fmax radix (Fmax radix ll lu) (Fmax radix ul uu)) in
 (andb
  (evaluate_even_closest radix bound precision l (value zl))
  (evaluate_even_closest radix bound precision u (value zu))).

Local min_mult := [xl,xu,yl,yu:cFloat] (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:cFloat] (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_f2ec_bound:
 (xl,xu,yl,yu,zl,zu:cFloat) (xr,yr,zr:cFloat)
 (Eint_f2_bound xl xu xr) → (Eint_f2_bound yl yu yr) →
 (cFloat_mult xr yr zr) →
 (Emult_f2ec_bound_helper xl xu yl yu zl zu) = true →
 (Eint_f2_bound zl zu zr).

Definition Emult_f2ec_error_helper :=
 [xe,ye,ze:float] [xn,yn:float] [zl,zu:cFloat]
 (Fle_bool radix
   (Fplus radix (Fplus radix
        (Fplus radix (Fmult xe yn) (Fmult ye xn))
        (Fmult xe ye))
      (cFloat_error zl zu)) ze).

Lemma Emult_f2ec_error_aux:
 (xe,ye,ze:float) (xn,yn,zl,zu:cFloat) (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2_error xe xr xx) → (Eint_f2_error ye yr yy) →
 (cFloat_mult xr yr zr) → (Eint_f2_bound zl zu zr) →
 "(Rabsolu xr) ≤ xn" → "(Rabsolu yr) ≤ yn" →
 (Emult_f2ec_error_helper xe ye ze (value xn) (value yn) zl zu) = true →
 (Eint_f2_error ze zr "xx × yy").

Lemma Emult_f2ec_error:
 (xl,xu:cFloat;xe:float) (yl,yu:cFloat;ye:float) (zl,zu:cFloat;ze:float)
 (xr,yr,zr:cFloat) (xx,yy:R)
 (Eint_f2 xl xu xe xr xx) → (Eint_f2 yl yu ye yr yy) →
 (cFloat_mult xr yr zr) → (Eint_f2_bound zl zu zr) →
 (Emult_f2ec_error_helper xe ye ze (Fmag radix (value xl) (value xu))
  (Fmag radix (value yl) (value yu)) zl zu) = true →
 (Eint_f2_error ze zr "xx × yy").

Axiom Ediv_f2ec_bound:
```

(*xl*,*xu*,*yl*,*yu*,*zl*,*zu*:*cFloat*) (*xr*,*yr*,*zr*:*cFloat*)
(*Eint_f2_bound xl xu xr*) → (*Eint_f2_bound yl yu yr*) →
(*cFloat_div xr yr zr*) →
true = *true* →
(*Eint_f2_bound zl zu zr*).

Axiom *Ediv_f2ec_error*:
(*xl*,*xu*:*cFloat*;*xe*:*float*) (*yl*,*yu*:*cFloat*;*ye*:*float*) (*zl*,*zu*:*cFloat*;*ze*:*float*)
(*xr*,*yr*,*zr*:*cFloat*) (*xx*,*yy*:*R*)
(*Eint_f2 xl xu xe xr xx*) → (*Eint_f2 yl yu ye yr yy*) →
(*cFloat_div xr yr zr*) → (*Eint_f2_bound zl zu zr*) →
true = *true* →
(*Eint_f2_error ze zr* "*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 *precisionMoreThanOne*: (*lt* (1) *precision*).
Unfold precision.
Auto with zarith.
Qed.
Local *bExp* := '*N*:149'.

Definition *Eint_s754* := (*Eint_f2 precision bExp*).
Definition *Eint_s754_bound* := (*Eint_f2_bound precision bExp*).
Definition *Eint_s754_error* := (*Eint_f2_error precision bExp*).
Definition *Eint_s754_union* := (*Eint_f2_union precision precisionMoreThanOne bExp*).
Definition *Eint_s754_inclusion* := (*Eint_f2_inclusion precision bExp*).

Definition *Eplus_s754ec_error* := (*Eplus_f2ec_error precision precisionMoreThanOne bExp*).
Definition *Eplus_s754ec_bound* := (*Eplus_f2ec_bound precision precisionMoreThanOne bExp*).
Definition *Eminus_s754ec_error* := (*Eminus_f2ec_error precision precisionMoreThanOne bExp*).
Definition *Eminus_s754ec_bound* := (*Eminus_f2ec_bound precision precisionMoreThanOne bExp*).
Definition *Emult_s754ec_bound* := (*Emult_f2ec_bound precision precisionMoreThanOne bExp*).
Definition *Emult_s754ec_error* := (*Emult_f2ec_error precision precisionMoreThanOne bExp*).
Definition *Ediv_s754ec_bound* := (*Ediv_f2ec_bound precision bExp*).
Definition *Ediv_s754ec_error* := (*Ediv_f2ec_error precision bExp*).
Definition *Eplus_exact_s754ec_error* := (*Eplus_exact_f2ec_error precision precisionMoreThanOne bExp*).
Definition *Eplus_exact_s754ec_bound* := (*Eplus_exact_f2ec_bound precision precisionMoreThanOne bExp*).
Definition *Eminus_exact_s754ec_error* := (*Eminus_exact_f2ec_error precision precisionMoreThanOne bExp*).
Definition *Eminus_exact_s754ec_bound* := (*Eminus_exact_f2ec_bound precision precisionMoreThanOne bExp*).

Definition *float_s754* := (*cFloat precision bExp*).
Definition *Good_s754* := (*Fcanonic radix* (*bound precision bExp*)).
Definition *Float_s754* := [*f*:*float*;*p*:(*Good_s754 f*)] (*CFloat precision bExp f p*).
Definition *Plus_s754ec* := (*cFloat_plus precision bExp*).
Definition *Minus_s754ec* := (*cFloat_minus precision bExp*).
Definition *Mult_s754ec* := (*cFloat_mult precision bExp*).
Definition *Div_s754ec* := (*cFloat_div precision bExp*).

End *Efloat_s754ec*.

Module IbaseR

Require Import *Reals*.

Section *IbaseR*.

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

Lemma *IplusR* :
∀ *xl xu yl yu zl zu x y* : *R*,
(*zl* ≤ *xl* + *yl*)%*R* → (*xu* + *yu* ≤ *zu*)%*R* →
Iint xl xu x → *Iint yl yu y* →
Iint zl zu (*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*,
Iint xl xu x → *Iint yl yu y* →
let (*zl*,*zu*) := *IplusR_fun* (*xl*,*xu*) (*yl*,*yu*) *in*
Iint zl zu (*x* + *y*).
intros xl xu yl yu x y Ix Iy.
simpl.
apply IplusR with xl xu yl yu; *auto with real*.
Qed.

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

Lemma *IoppR_exact_rev* :
∀ *xl xu x* : *R*, *Iint* (- *xu*) (- *xl*) (- *x*) → *Iint 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 *ImultR_pp* :
∀ *xl xu yl yu zl zu x y* : *R*,
(0 ≤ *xl*)%*R* → (0 ≤ *yl*)%*R* →
(*zl* ≤ *xl* × *yl*)%*R* → (*xu* × *yu* ≤ *zu*)%*R* →
Iint xl xu x → *Iint yl yu y* →
Iint zl zu (*x* × *y*).

Lemma *ImultR_pm* :
∀ *xl xu yl yu zl zu x y* : *R*,
(0 ≤ *xl*)%*R* → (*yl* ≤ 0)%*R* → (0 ≤ *yu*)%*R* →
(*zl* ≤ *xu* × *yl*)%*R* → (*xu* × *yu* ≤ *zu*)%*R* →
Iint xl xu x → *Iint yl yu y* →
Iint zl zu (*x* × *y*).

Lemma *ImultR_pn* :
∀ *xl xu yl yu zl zu x y* : *R*,
(0 ≤ *xl*)%*R* → (*yu* ≤ 0)%*R* →

$(zl \leq xu \times yl)\%R \rightarrow (xl \times yu \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x \times y).$

Lemma $ImultR_mp$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(xl \leq 0)\%R \rightarrow$
$(0 \leq xu)\%R \rightarrow$
$(0 \leq yl)\%R \rightarrow$
$(zl \leq xl \times yu)\%R \rightarrow$
$(xu \times yu \leq zu)\%R \rightarrow Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow Iint\ zl\ zu\ (x \times y).$

Lemma $ImultR_mm$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(xl \leq 0)\%R \rightarrow (0 \leq xu)\%R \rightarrow (yl \leq 0)\%R \rightarrow (0 \leq yu)\%R \rightarrow$
$(zl \leq xu \times yl)\%R \rightarrow (zl \leq xl \times yu)\%R \rightarrow$
$(xl \times yl \leq zu)\%R \rightarrow (xu \times yu \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x \times y).$

Lemma $ImultR_mn$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(xl \leq 0)\%R \rightarrow (0 \leq xu)\%R \rightarrow (yu \leq 0)\%R \rightarrow$
$(zl \leq xu \times yl)\%R \rightarrow (xl \times yl \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x \times y).$

Lemma $ImultR_np$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(xu \leq 0)\%R \rightarrow (0 \leq yl)\%R \rightarrow$
$(zl \leq xl \times yu)\%R \rightarrow (xu \times yl \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x \times y).$

Lemma $ImultR_nm$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(xu \leq 0)\%R \rightarrow (yl \leq 0)\%R \rightarrow (0 \leq yu)\%R \rightarrow$
$(zl \leq xl \times yu)\%R \rightarrow (xl \times yl \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x \times y).$

Lemma $ImultR_nn$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(xu \leq 0)\%R \rightarrow (yu \leq 0)\%R \rightarrow$
$(zl \leq xu \times yu)\%R \rightarrow (xl \times yl \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x \times y).$

Inductive $interval_sign\ (l\ u : R) : Set :=$
$\quad | \ Nsign : (u \leq 0)\%R \rightarrow interval_sign\ l\ u$
$\quad | \ Msign : (l \leq 0)\%R \rightarrow (0 \leq u)\%R \rightarrow interval_sign\ l\ u$
$\quad | \ Psign : (0 \leq l)\%R \rightarrow interval_sign\ l\ u.$

Lemma $interval_sign_correct : \forall\ l\ u : R,\ (l \leq u)\%R \rightarrow interval_sign\ l\ u.$

Lemma $Iint_sign : \forall\ l\ u\ x : R,\ Iint\ l\ u\ x \rightarrow interval_sign\ l\ u.$

Lemma $Rmin_trans$:
$\forall\ a\ b\ c : R,\ (c \leq Rmin\ a\ b)\%R \rightarrow (c \leq a)\%R \wedge (c \leq b)\%R.$

Lemma $Rmax_trans$:
$\forall\ a\ b\ c : R,\ (Rmax\ a\ b \leq c)\%R \rightarrow (a \leq c)\%R \wedge (b \leq c)\%R.$

Lemma $ImultR$:

$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(zl \leq Rmin\ (Rmin\ (xl \times yl)\ (xl \times yu))\ (Rmin\ (xu \times yl)\ (xu \times yu)))\%R \rightarrow$
$(Rmax\ (Rmax\ (xl \times yl)\ (xl \times yu))\ (Rmax\ (xu \times yl)\ (xu \times yu)) \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow Iint\ zl\ zu\ (x \times y).$

Definition $ImultR_fun\ (xi\ yi : R \times R) :=$
$let\ (xl,xu) := xi\ in\ let\ (yl,yu) := yi\ in$
$((Rmin\ (Rmin\ (xl \times yl)\ (xl \times yu))\ (Rmin\ (xu \times yl)\ (xu \times yu)))\%R,$
$(Rmax\ (Rmax\ (xl \times yl)\ (xl \times yu))\ (Rmax\ (xu \times yl)\ (xu \times yu)))\%R).$

Lemma $ImultR_fun_correct$:
$\forall\ xl\ xu\ yl\ yu\ x\ y : R,$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$let\ (zl,zu) := ImultR_fun\ (xl,xu)\ (yl,yu)\ in$
$Iint\ zl\ zu\ (x \times y).$
intros xl xu yl yu x y Ix Iy.
simpl.
apply ImultR with xl xu yl yu; auto with real.
Qed.

Lemma $Rle_Rinv : \forall\ x\ y : R, (0 < x)\%R \rightarrow (x \leq y)\%R \rightarrow (/\ y \leq /\ x)\%R.$

Lemma $IinvR_p$:
$\forall\ yl\ yu\ zl\ zu\ y : R,$
$(0 < yl)\%R \rightarrow$
$Iint\ yl\ yu\ y \rightarrow (zl \leq /\ yu)\%R \rightarrow (/\ yl \leq zu)\%R \rightarrow Iint\ zl\ zu\ (/\ y).$

Lemma $IinvR_n$:
$\forall\ yl\ yu\ zl\ zu\ y : R,$
$(yu < 0)\%R \rightarrow$
$Iint\ yl\ yu\ y \rightarrow (zl \leq /\ yu)\%R \rightarrow (/\ yl \leq zu)\%R \rightarrow Iint\ zl\ zu\ (/\ y).$

Lemma $IinvR$:
$\forall\ yl\ yu\ zl\ zu\ y : R,$
$(yu < 0)\%R \vee (0 < yl)\%R \rightarrow$
$Iint\ yl\ yu\ y \rightarrow (zl \leq /\ yu)\%R \rightarrow (/\ yl \leq zu)\%R \rightarrow Iint\ zl\ zu\ (/\ y).$

Lemma $IdivR$:
$\forall\ xl\ xu\ yl\ yu\ zl\ zu\ x\ y : R,$
$(yu < 0)\%R \vee (yl > 0)\%R \rightarrow$
$(zl \leq Rmin\ (Rmin\ (xl\ /\ yu)\ (xl\ /\ yl))\ (Rmin\ (xu\ /\ yu)\ (xu\ /\ yl)))\%R \rightarrow$
$(Rmax\ (Rmax\ (xl\ /\ yu)\ (xl\ /\ yl))\ (Rmax\ (xu\ /\ yu)\ (xu\ /\ yl)) \leq zu)\%R \rightarrow$
$Iint\ xl\ xu\ x \rightarrow Iint\ yl\ yu\ y \rightarrow$
$Iint\ zl\ zu\ (x\ /\ y).$

End *IbaseR*.