Generating and Certifying Accuracy Properties of Floating-Point Programs

Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché

Generating and Certifying Accuracy Properties of Floating-Point Programs

Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché

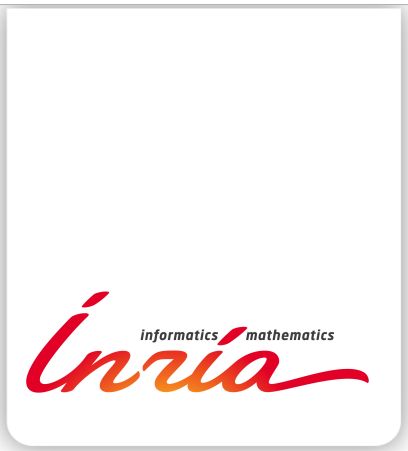
RESEARCH REPORT

N° 9564

December 2024

Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--9564--FR+ENG



Generating and Certifying Accuracy Properties of Floating-Point Programs *

Paul Bonnot†, Benoît Boyer‡, Florian Faissole‡, Claude Marché†

Project-Team Toccata

Research Report n° 9564 — December 2024 — 41 pages

Abstract: Numerical programs make use of the floating-point representation of numbers to perform computations that ideally should be done on mathematical real numbers. The floating-point representation induces approximations on the computations ultimately performed. The accuracy property of such a numerical program is expressed as a mathematical formula about the difference between the computed result and the ideal computation. In this work, we bound this difference by a formula involving the assumed precision of the input of the programs, together with the accuracy properties of the auxiliary mathematical functions that are called as basic blocks. Obtaining such an accuracy property needs a high expertise in floating-point computer arithmetic.
We propose a methodology that is able to automatically generate such form of accuracy formulas from a given input program. Moreover the generated formulas are certified correct with a high level of confidence, thanks to the automated construction of formal proofs of their validity.
Our methodology is implemented and experimented on several examples involving approximations of elementary functions such as sine, cosine, exponential and logarithm.

Key-words: Formal Specification, Deductive Verification, Why3 Environment for Deductive Verification, Floating-Point Representation, Numerical Programs.

* This work has been partly supported by the bilateral contract ProofInUse-MERCE between Inria team Toccata and Mitsubishi Electric R&D Centre Europe, Rennes ; and partly by the Décysif project funded by the Île-de-France region and by the French government in the context of "Plan France 2030"

† Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France

‡ Mitsubishi Electric R&D Centre Europe, Rennes, France

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Génération et certification de propriétés de précision de calculs effectués par des programmes en virgule flottante

Résumé : Les programmes numériques utilisent la représentation en virgule flottante des nombres pour effectuer des calculs qui devraient idéalement être effectués sur des nombres réels mathématiques. La représentation en virgule flottante induit des approximations sur les calculs réalisés. La propriété de précision d'un tel programme numérique est exprimée sous la forme d'une formule mathématique portant sur la différence entre le résultat calculé et le calcul idéal. Dans ce travail, nous bornons cette différence par une formule dépendant de la précision supposée des entrées du programme, ainsi que les propriétés de précision des fonctions mathématiques auxiliaires appelées comme blocs de base. L'obtention d'une telle propriété de précision nécessite une grande expertise en arithmétique à virgule flottante.

Nous proposons une méthodologie capable de générer automatiquement, à partir de programmes utilisateurs, des formules exprimant leur degré de précision. De plus, les formules générées sont certifiées correctes avec un niveau de confiance élevé, grâce à la construction automatisée de preuves formelles de leur validité.

Notre méthodologie est mise en œuvre et expérimentée sur plusieurs exemples impliquant des approximations de fonctions élémentaires telles que le sinus, le cosinus, l'exponentielle et le logarithme.

Mots-clés : Spécification formelle, preuve de programmes, environnement Why3 pour la vérification déductive, Nombres en virgule flottante, Programmes numériques.

Ces recherches ont été en partie financées par le contrat bilatéral ProofInUse-MERCE entre l'équipe Inria Toccata et Mitsubishi Electric R&D Centre Europe, à Rennes ; et en partie par le projet Décysif financé par la région Île-de-France et par le gouvernement français dans le cadre du « Plan France 2030 »

Contents

List of Figures

1 Introduction

Numerical computations are ubiquitous in many industrial systems software. They are typically based on floating-point arithmetic, which is standardised by the IEEE-754 standard [27]. The *Handbook of Floating-Point Arithmetic* [33] presents a wide overview of the topic of computer arithmetic.

The objective of the research presented in this document is to analyse the *accuracy* of numerical programs. In this context, by accuracy we mean the precision of the obtained result of computation with respect to an ideal computation that would be made on mathematical real numbers: the closer the two results are, the better, that is the more accurate is the program.

The precision of the computation is of course dependent of some unknown input parameters. Our objective is not to determine the accuracy for particular concrete values of these parameters, but to generate a closed formula expressing that accuracy, valid for any values of the parameters from given domains. As a toy example, assume we want to compute the mathematical expression $r\cos\theta$ for some non-negative radius r, say smaller than 1, and some angle θ, say between $-\pi$ and π. The corresponding computer program, in the C language and using single precision floating-point numbers, looks like the following.

```
#include <math.h>
float myprog(float r, float theta) {
  return r * cosf(theta);
}
```

This program invokes the floating-point multiplication, for which precision can be assumed to respect the IEEE-754 standard, and also the `cosf` function from some library, whose precision is not standardised. Not only these operations introduce approximations, but we may also assume that the parameters r and θ are themselves some approximations of real inputs $\bar{r}$ and $\bar{\theta}$. So the question is: can we provide a closed formula relating the accuracy of the result to the respective accuracy of the approximations of r, θ, and the precision of the `cosf` function. As a teaser for the rest of this article, our methodology allows to discover, in a mostly automated manner, the formula

$$\begin{array}{rcl} |\mathsf{result} - \bar{r}\cos\bar{\theta}| & \leq & (\varepsilon + E^{\mathrm{rel}}_{\cos}(1+\varepsilon))|\bar{r}\cos\bar{\theta}| \\ & + & (t\bar{r} + (r_{abs} + r_{abs}E^{\mathrm{rel}}_{\cos})|\cos\bar{\theta}| + r_{abs}t) \times (1+\varepsilon) + \eta \end{array} \tag{1}$$

where $t = \theta_{abs}(1+E^{\mathrm{rel}}_{\cos}) + E^{\mathrm{abs}}_{\cos}$. In the formula above, result denotes the returned value of the call to $\mathtt{myprog}(r,\theta)$. The term $\varepsilon + E^{\mathrm{rel}}_{\cos}(1+\varepsilon)$ above, in factor of $|\bar{r}\cos\bar{\theta}|$ is called the relative error, where the rest of the formula on the second line is called the absolute error. In these formulas, $\bar{r}$ is a real of which r is a floating-point approximation, with $|r-\bar{r}| \leq r_{abs}$, and $\bar{\theta}$ is a real of which θ is a floating-point approximation, with $|\theta - \bar{\theta}| \leq \theta_{abs}$. The parameters ε and η are characteristic of the floating-point format used (see Section 2.1) and finally, $E^{\mathrm{rel}}_{\cos}$ and $E^{\mathrm{abs}}_{\cos}$ are parameters for which we assume that for any float x and real $\bar{x}$ between $-\pi$ and π:

$$|\mathtt{cosf}(x) - \cos(\bar{x})| \leq E^{\mathrm{rel}}_{\cos}|x-\bar{x}| + E^{\mathrm{abs}}_{\cos}$$

The values for $E^{\mathrm{rel}}_{\cos}$ and $E^{\mathrm{abs}}_{\cos}$ depend on which library is used for computing the cosine: implementations that are commonly provided with C compilers do not guarantee the best precision. The best possible precision for given a floating-format is only guaranteed by specialised libraries such as CR-LIBM [14] and CORE-MATH [36]. This comes with a cost though, as the best implementations take more time to compute. In contexts such as IoT, lower precision implementations with lower costs are perfectly acceptable and indeed better. As seen in the example above, in this work, the formulas we generate are parametric in the precision of the called functions.

Even on a very simple program as above, an accuracy formula taking into account both the accuracy of the input parameters and the parametric accuracy of library functions like `cosf` becomes very quickly

complex, so some automation in generating such formulas is desirable. The analysis of accuracy we seek to perform here is in fact two-fold. On the first hand, we want to automatically generate accuracy formulas as in the example above. On the other hand, we want a very high level of guarantee, so we seek for a *formal proof* of the validity of the generated formula. Formally proving the accuracy of floating-point computations is a complex topic addressed by different approaches in the scientific literature. Recent overviews of this topic can be found in Melquiond's "Habilitation" dissertation [32] and a survey by Boldo *et al.* [6]. In the present work, we want to obtain the formal proofs in a mostly automatic manner, because we aim to apply this approach to many different programs. A too-much interactive proof, that would need to be manually updated for each simple change of the input program, is not an option. The need for automation is in particular motivated by examples considered in a previous work of ours [7, 8], these examples being considered again in this report.

The structure of the report is as follows. In Section 2 we start to expose a few preliminaries, detailing in particular the general form of accuracy formulas we seek, and the way we plan to obtain them automatically using a so-called *strategy* within the Why3 environment for deductive verification. Section 3 presents the core of an algorithm for generating accuracy formulas together with formal proofs generated at the same time. It is illustrated on the addition operator only. Section 4 extends the core algorithm to subtraction and multiplication. In Section 5 we add the support for library functions, illustrated on the case of the sine and cosine trigonometric functions. Section 6 extends again the core algorithm with support for exponential and logarithm. We conclude and discuss related work and future work in Section 7.

Notice that the source code of the strategy for generating accuracy formulas, and all the examples of this report, are present in the master branch of the repository of Why3 (`https://gitlab.inria.fr/why3/why3`) and will appear in its upcoming release 1.8.0 (see `https://www.why3.org/`). In details, the propagation lemma are stated and proved in the file `stdlib/ufloat.mlw` and the examples are implemented and proved in the directory `examples/numeric`.

2 Preliminaries

We present in this section a few preliminaries. In Section 2.1 we summarise the essentials of floating-point arithmetic. Section 2.2 gives a overview of the Why3 environment and provides a focus on the methodology of proof strategies in that context. In Section 2.3 we review the concept of *unbounded* floating-point numbers, and how they are modelled in Why3. Section 2.4 gives a general overview of the methodology that we develop in further sections.

2.1 Floating-Point Arithmetic in Short

The IEEE-754 standard [27] defines several formats of representation of floating-point numbers. A binary format is characterised by a precision p as well as upper and lower bounds $e_{\max}$ and $e_{\min}$ for the exponent. A floating point number is either a value among $+\infty$, $-\infty$ and NaN or a value $\pm m \times 2^{e-p+1}$ where $0 \leq m \leq 2^p - 1$, $e_{\min} \leq e \leq e_{\max}$. The largest representable number is this format is $\mathsf{maxfloat} = (2 - 2^{-p-1}) \times 2^{e_{\max}}$, and the smallest positive representable number is $2^{e_{\min}-p+1}$.

In this report we are not interested in a particular format since our proof methodology is independent of the format used, however only 2 formats are currently supported in our formal proofs :

- Single format (32 bits, `float` in C) where $p = 24$, $e_{\max} = 127$ and $e_{\min} = -126$
- Double format (64 bits, `double` in C) where $p = 53$, $e_{\max} = 1023$ and $e_{\min} = -1022$

We use the symbol rnd to denote the *rounding* of a real number to a floating-point number. The IEEE-754 standard defines several rounding modes. In this report we consider only the mode *nearest-ties-to-even*: when a real number x lies within an interval $[x_1; x_2]$ of two consecutive floating-point numbers, then

$\mathsf{rnd}(x)$ is either x_1 or x_2: the one of these which is closest to x, or in case x is exactly in the middle, the one among x_1 and x_2 whose mantissa is even. Also, when x is too large (larger than or equal to the middle of $\mathsf{maxfloat}$ and $2^{e_{\max}}$), $\mathsf{rnd}(x)$ is $+\infty$. We use the symbols $\oplus$, $\ominus$, $\otimes$, $\oslash$ to denote the basic operations of addition, subtraction, multiplication and division of floating-point numbers. As specified by IEEE-754, all these operations must use the best possible rounding, that is $x \oplus y = \mathsf{rnd}(x+y)$ and similarly for the three other operations.

The main property of the rounding function that we use in this report is the following: for any real number x such that $|x| \leq \mathsf{maxfloat}$, $\mathsf{rnd}(x)$ is finite and

$$|\mathsf{rnd}(x) - x| \leq \varepsilon|x| + \eta \tag{2}$$

where $\varepsilon = \frac{2^{-p}}{1+2^{-p}}$ and $\eta = 2^{e_{\min}-p}$. This property can be considered as well-known and folklore in the literature, see for example Jeannerod and Rump [28]. In seminal publications, such as the Handbook of Computer Arithmetic [33] or Higham's survey [26], the simpler term $\varepsilon = 2^{-p}$ is used instead of $\frac{2^{-p}}{1+2^{-p}}$, inducing a slightly larger bound. Jeannerod and Rump [28, Theorem 2.1] showed that the refined bound is actually optimal in the sense that there exist some inputs values and floating-point formats (with certain conditions) for which it is attained. In most cases, the precision gain obtained using this optimal bound instead of 2^{-p} is small. Anyway, the latter results and proofs simply use the symbol ε to denote either of the bounds.

As remarked by Jeannerod and Rump [28], Property 2 can be refined in the special case of addition because underflowing additions are exact:

$$|(x \oplus y) - (x+y)| \leq \varepsilon|x+y| \tag{3}$$

that is, the term η can be removed from formula 2. Moreover, it should be noted that (see for example the Handbook of Floating-Point Arithmetic [33])

$$|(x \oplus y) - (x+y)| \leq |x| \tag{4}$$

and symmetrically

$$|(x \oplus y) - (x+y)| \leq |y| \tag{5}$$

The combination of the formulas 3, 4 and 5 is used later on to obtain better bounds on additions, and also on compound sums.

2.2 The Why3 Environment in Short

Why3 is a generic environment for deductive program verification (see `http://www.why3.org`), providing the language WhyML for specification and programming [20, 5, 21]. The genericity of Why3 is exemplified by the fact that WhyML is already used as an intermediate language for verification of programs written in C, Java, Ada [19, 29] and Rust [18]. The specification component of WhyML [4], used to write program annotations and background theories, is an extension of first-order logic. The specification part of the language serves as a common format for theorem proving problems, *proof tasks* in Why3's jargon. Why3 generates proof tasks from user lemmas and annotated programs, using a weakest-precondition calculus, then dispatches them to multiple provers. It is indeed another aspect of the genericity of Why3: its ability to dispatch proof tasks to many different provers. In practice, for the proof of programs, provers of the SMT (*Satisfiability Modulo Theories*) family are the most successful ones, indeed at least if they support quantified formulas, which is the case for Alt-Ergo [12], CVC4 [3], cvc5 [2] and Z3 [17].

2.2.1 Floating-Point Numbers in Why3

When working with the Why3 environment, one can make use of floating-point operations via the standard library, which provides a module for floating-point numbers that is faithful to the IEEE-754 standard. Type declarations for floating-point numbers are provided. This includes the types named `single` and `double` for the 32-bit format and 64-bit format respectively. That library provides operations including `rnd` (denoted `round` in that library), and also operations $\oplus$, $\ominus$, etc.

This IEEE-compliant float library is used by some front-ends of Why3, in particular the SPARK environment for verification of Ada programs, and the J3 plugin of TIS-Analyzer for C code. Fumex et al. [22, 23] made use of that library to verify a few complex numerical programs in Ada. This library is designed in order to exploit the support for floating-point arithmetic provided by SMT solvers [9]. This means that reasoning is done at the level of the bit representation of floating-point numbers. This appears to be reasonably efficient for reasoning on properties expressed directly at the level of floats, *e.g.* the absence of overflow. On the contrary, if one wants to express and prove properties that relate floating-point computations to real numbers, it fails completely: the provers are hardly able to mix reasoning of reals and on floats, in particular do not have knowledge of the properties (2)-(5) above. As a consequence, our proof strategy will not use at all the provers' capabilities regarding IEEE floating-point numbers, but only reason with real numbers. This is detailed further in Section 2.3 and Section 2.4 below.

2.2.2 Proof Strategies in Why3

Besides the ability to resolve proof tasks by calling external provers, Why3 provide a notion of *transformations* [25] that can be invoked to decompose or simplify task into sub-tasks, in many different ways. This notion is very similar to the notion of tactics available in interactive proof assistant such as Coq and Isabelle/HOL. Transformations can be invoked manually in the Why3 IDE, but these manual invocations are typically for quite expert users. Why3 also provide a notion of *strategy* for making this use more automatic.

To automate the proofs of numerical programs, we make an advanced use of Why3 strategies. In fact, we made a new technical contribution to Why3, so as to propose the so-called *task-oriented* strategies. Such a strategy is a procedure implemented directly in OCaml (the implementation language of Why3) that takes a proof task as argument, and produces as output some form of a *proof tree*: such a proof tree describes a sequence of transformations to be executed in order to prove the task. It has a tree shape because the transformations can themselves generate subtasks with their own proof trees. More generally, such a strategy can generate a *partial* proof tree, leaving some sub-tasks unproved: therefore, task-oriented strategies can be used before dispatching remaining sub-tasks to SMT solvers. This helps the solvers to obtain proofs, by generating intermediary steps with transformations. Task-oriented strategies can also be used to select which solvers to run depending on the goal. For instance, if the goal is a numerical goal, a goal-oriented strategy might choose to use a specific prover specialized in these types of proofs. That's why the *leaves* of such proof trees are typically a call to one or several provers.

Here is the declaration of an algebraic data type for partial proof trees

```
type proof_tree =
  | Sapply_trans(string, string list, proof_tree list)
  | Scall_prover(prover list)
```

The constructor `Sapply_trans`(n,a,t) represents the invocation of the Why3 transformation named n with arguments a that will be applied to the current task. The last parameter t is the list of proof trees that should be recursively applied to the sub-tasks generated by the transformation. The constructor `Scall_prover`(P) indicates a list of provers P that should be invoked to attempt to solve the goal (or, recursively, sub-goals). Notice that the proof tree `(Scall_prover [])` is a way to tell to do nothing at all.

It is important to notice that such a proof tree is not a proof per se, it is the skeleton of a proof. When applying a Why3 task-oriented strategy, such a proof tree is first generated, and then in a second step the Why3 core must apply the proof tree to the initial task. It is perfectly possible that such an application fail on some sub-task: in that case, the sub-task will be left as is as one of a sub-goal remaining to prove. This mechanism in two-steps, generating a proof tree first and then somehow running it, has a very important advantage: the implementation of the strategy does not need to be trusted. The trust on the proof only relies on the Why3 transformation invoked, and the provers. If a strategy is not correct, in the sense that the generated proof tree is not a correct proof, the second step will not complete, and will leave the unproved tasks to prove.

2.3 Unbounded Floating-Point Numbers

As said at the end of Section 2.2.1, we want to reason using Properties (2)- (5). There is a difficulty though: these properties are in fact not always valid, but only when the underlying operations rnd, $\oplus$, etc. *do not overflow*. Otherwise, the result of these operations are $\pm\infty$, as specified by the IEEE standard. This situation is annoying for proofs, because it means we constantly have to take care if overflow does occur or not. In fact, in the literature on floating-point arithmetic, it is common practice to reason, at least on paper, without taking care of overflows, and only justify absence of overflows separately. This is not appropriate when doing formal proofs. For this reason, we are going to consider the so-called *unbounded floating-point numbers*, as we already did in a previous work of ours [7, 8].

Roughly speaking, the set of unbounded floating-point numbers is an extension of the IEEE floating-point numbers described in Section 2.1, except that there is no maximal exponent $e_{\max}$ anymore. On that extended set, there is no possible overflow anymore, and thus no need for special values $\pm\infty$ and NaN (let us ignore division here). The properties 2, 3, 4 and 5 indeed hold for unbounded floating-point numbers.

Using unbounded floating-point numbers with Why3 has already been made available as a new Why3 theory [7, 8]. This library provides the types `usingle` and `udouble` for unbounded floats in 32-bit or 64-bit formats. For each of these types there is a function `to_real` giving the real represented by its argument. In the remaining of this paper, we will often omit the `to_real` function to improve readability, so that at the end we can state formulas about unbounded floating-point numbers on paper as if we were stating formulas on real numbers. More details about the Why3 theory of unbounded floating-point numbers can be found in Bonnot *et al* [7].

2.4 Forward Error Formulas and their Propagation

A *forward error formula* is a formula that relates some unbounded floating-point number $\hat{x}$ and a real number x. It is meant to express that $\hat{x}$ is an approximation of x, and the formula in question expresses a property on the accuracy of this approximation. Such a forward error formula has the form[1]

$$|\hat{x} - x| \leq x_{rel}\tilde{x} + x_{abs}$$

where x_{rel}, $\tilde{x}$ and x_{abs} are non-negative real numbers respectively called the *relative error*, the *factor* and the *absolute error*. There is no formal constraint on what is the factor, although it is expected to be something like the absolute value of x, or something slightly larger, as seen in examples in the rest of this report. A first example of a propagation error formula is given by Formula (1) of our introductory example, where $(\varepsilon + E^{\text{rel}}_{\cos}(1+\varepsilon))$ is the relative error, $|\bar{r}\cos\bar{\theta}|$ is the factor, and $(t\bar{r} + (r_{abs} + r_{abs}E^{\text{rel}}_{\cos})|\cos\bar{\theta}| + r_{abs}t) \times (1+\varepsilon) + \eta$ is the absolute error. Another example of propagation error formulas are given by Property (2), where $\mathsf{rnd}(x)$ is seen as an approximation of x with relative error ε, factor $|x|$ and absolute error η. Another example is given by Property (3), where $x \oplus y$ is seen as an approximation of $x+y$ with relative error ε, factor $|x+y|$ and absolute error 0.

[1] As mentioned beforewe write $\hat{x}$ instead of `to_real` $\hat{x}$ for readability.

The *propagation* of forward errors aims at deriving, from a known forward error for some $\hat{x}$ approximating x, a forward error for $f(\hat{x})$, for some (unbounded) floating-point operation f. To be precise, we talk about the propagation of errors through f. Such a propagation should be a recipe to derive a real expression for which $f(\hat{x})$ is naturally an approximation, together with a new relative error, a new factor and a new absolute error. When that recipe is a non-ambiguous computation, we will talk about some *automated* propagation technique.

This concept of propagation generalizes naturally to propagation through a function f with several arguments. In that case the goal is to derive a forward error formula for $f(\hat{x_1},\ldots,\hat{x_n})$ from known forward error formulas for each $\hat{x_i}$.

Once a propagation method is defined precisely, our goal is to turn it into an task-oriented strategy within Why3. We start doing that in the next section which focuses on propagation through the unbounded floating-point addition, before extending the method to other operations in the remaining sections.

3 Forward Error Propagation: Case of Floating-Point Addition

This section starts the construction of a general algorithm for error propagation. It is initially presented for propagation through addition, and it will be extended further in the next sections.

We start in Section 3.1 by stating and proving a lemma on error propagation through addition, which is the basis for a propagation method. We then describe in Section 3.2 the general shape of the propagation algorithm. We present examples in Section 3.3.

3.1 Propagation Lemma for Addition

Lemma 3.1 (Error propagation through an addition)**.** *For any real numbers $x,y,\tilde{x},\tilde{y},x_{rel},y_{rel},x_{abs},y_{abs}$, and for any floats $\hat{x}$ and $\hat{y}$ (supposed to denote approximations of x and y, respectively), such that:*

- $|\hat{x}-x| \leq x_{rel}\tilde{x}+x_{abs}$
- $|\hat{y}-y| \leq y_{rel}\tilde{y}+y_{abs}$
- $|x| \leq \tilde{x}$
- $|y| \leq \tilde{y}$
- $x_{rel},y_{rel},x_{abs},y_{abs} \geq 0$

we have

$$|(\hat{x}\oplus\hat{y})-(x+y)| \leq (x_{rel}+y_{rel}+\varepsilon)(\tilde{x}+\tilde{y})+x_{abs}(1+\varepsilon+y_{rel})+y_{abs}(1+\varepsilon+x_{rel})$$

Proof. We first prove the following inequality:

$$|(\hat{x}\oplus\hat{y})-(\hat{x}+\hat{y})| \leq (\varepsilon+y_{rel})\tilde{x}+(\varepsilon+x_{rel})\tilde{y}+y_{abs}(x_{rel}+\varepsilon)+x_{abs}(y_{rel}+\varepsilon) \tag{6}$$

We distinguish three cases:

- if $\tilde{x}+x_{abs} \leq \varepsilon(\tilde{y}+y_{abs})$ then

$$\begin{aligned}
|(\hat{x}\oplus\hat{y})-(\hat{x}+\hat{y})| &\leq |\hat{x}| && \text{(by Property (4))}\\
&\leq |\hat{x}-x|+|x| && \text{(by triangular inequality)}\\
&\leq x_{rel}\tilde{x}+x_{abs}+\tilde{x} && \text{(by lemma assumptions)}\\
&\leq x_{rel}(\varepsilon(\tilde{y}+y_{abs})-x_{abs})+\varepsilon(\tilde{y}+y_{abs})\\
&\leq x_{rel}\varepsilon(\tilde{y}+y_{abs})+\varepsilon(\tilde{y}+y_{abs})\\
&\leq (\varepsilon+x_{rel})(\tilde{y}+y_{abs}) && \text{(because } \varepsilon \leq 1\text{)}
\end{aligned}$$

which proves (6).

- symmetrically, if $\tilde{y}+y_{abs} \leq \varepsilon(\tilde{x}+x_{abs})$, we conclude by using Property (5) and the same reasoning as above.
- otherwise, we have $\varepsilon(\tilde{x}+x_{abs}) \leq \tilde{y}+y_{abs}$ and $\varepsilon(\tilde{y}+y_{abs}) \leq \tilde{x}+x_{abs}$. We have

$$\begin{aligned}
|(\hat{x}\oplus\hat{y})-(\hat{x}+\hat{y})| &\leq \varepsilon|\hat{x}+\hat{y}| && \text{(by Property (3))}\\
&\leq \varepsilon|(\hat{x}-x+x)+(\hat{y}-y+y)|\\
&\leq \varepsilon(|\hat{x}-x|+|x|+|\hat{y}-y|+|y|) && \text{(by triangular inequality)}\\
&\leq \varepsilon(x_{rel}\tilde{x}+x_{abs}+\tilde{x}+y_{rel}\tilde{y}+y_{abs}+\tilde{y}) && \text{(by lemma assumptions)}\\
&= \varepsilon x_{rel}\tilde{x}+\varepsilon y_{rel}\tilde{y}+\varepsilon(x_{abs}+y_{abs}+\tilde{x}+\tilde{y})\\
&= \varepsilon x_{rel}(\tilde{x}+x_{abs})+\varepsilon y_{rel}(\tilde{y}+y_{abs})+\varepsilon(x_{abs}+y_{abs}+\tilde{x}+\tilde{y})\\
&= x_{rel}(\tilde{y}+y_{abs})+y_{rel}(\tilde{x}+x_{abs})+\varepsilon(x_{abs}+y_{abs}+\tilde{x}+\tilde{y})\\
& && \text{(by this subcase assumption)}\\
&= (\varepsilon+y_{rel})\tilde{x}+(\varepsilon+x_{rel})\tilde{y}+y_{abs}(x_{rel}+\varepsilon)+x_{abs}(y_{rel}+\varepsilon)
\end{aligned}$$

We now use Property (6) to prove our lemma.

$$\begin{aligned}
\hat{x}\oplus\hat{y} &\leq (\hat{x}+\hat{y})+(\varepsilon+y_{rel})\tilde{x}+(\varepsilon+x_{rel})\tilde{y}+y_{abs}(x_{rel}+\varepsilon)+x_{abs}(y_{rel}+\varepsilon)\\
&\leq x+x_{rel}\tilde{x}+x_{abs}+y+y_{rel}\tilde{y}+y_{abs}+(\varepsilon+y_{rel})\tilde{x}+(\varepsilon+x_{rel})\tilde{y}+y_{abs}(x_{rel}+\varepsilon)+x_{abs}(y_{rel}+\varepsilon)\\
&= (x+y)+(x_{rel}+y_{rel}+\varepsilon)(\tilde{x}+\tilde{y})+x_{abs}(1+\varepsilon+y_{rel})+y_{abs}(1+\varepsilon+x_{rel})
\end{aligned}$$

and similarly

$$\begin{aligned}
\hat{x}\oplus\hat{y} &\geq (\hat{x}+\hat{y})-(\varepsilon+y_{rel})\tilde{x}-(\varepsilon+x_{rel})\tilde{y}-y_{abs}(x_{rel}+\varepsilon)-x_{abs}(y_{rel}+\varepsilon)\\
&\geq x-x_{rel}\tilde{x}-x_{abs}+y-y_{rel}\tilde{y}-y_{abs}-(\varepsilon+y_{rel})\tilde{x}-(\varepsilon+x_{rel})\tilde{y}-y_{abs}(x_{rel}+\varepsilon)-x_{abs}(y_{rel}+\varepsilon)\\
&= (x+y)-(x_{rel}+y_{rel}+\varepsilon)(\tilde{x}+\tilde{y})-x_{abs}(1+\varepsilon+y_{rel})-y_{abs}(1+\varepsilon+x_{rel})
\end{aligned}$$

□

3.2 An Algorithm for Automatic Generation of Forward Errors

The forward propagation algorithm is described incrementally in this paper. In this subsection, we describe how it operates with the addition lemma 3.1 as the only supported lemma. We will later on add other lemmas and augment the algorithm accordingly.

The main structure of the algorithm is presented on Figure 1. It manipulates data structures of type `fe` standing for "forward error", declared as the following record type.

```
type fe = { exact: term; rel: term; factor: term, abs: term }
```

The intention is that if for a term t we compute a forward error e of type `fe`, then we have inferred the formula

$$|t-e.exact| \leq e.rel \times e.factor + e.abs$$

Given a proof task and a floating-point term t, the function `forward_error` of Figure 1 tries to infer a forward error formula of the shape above, for the term t. Simultaneously, it computes a proof tree for that formula. This function proceeds in two phases. The first phase computes a table of assumed errors from the assumptions of the given proof task. This table is a finite mapping from floating-point terms to forward errors, i.e `fe` data structures. The second phase, corresponding to the function `generate` in

```
(* generates a forward error from a proof task and a given float term *)
function forward_error(Task:proof task,t:term) : proof_tree
  let Δ = map scan_hypot (hypotheses(Task)) in
  case generate(Δ,t) of
  | None: Scall_prover []  (* do nothing *)
  | Some(e,T): T
  end case

(* checks whether an hypothesis has the form of a forward error formula *)
function scan_hypot(f:fmla):
  case on the shape of f:
  | |to_real t − u| ≤ v:
    factorise v under the form a × |u| + b
    if a is non zero:
      return mapping t ↦ { exact = u; rel = a; factor = |u|; abs = b }
    else
      factorise v under the form a × u + b
      (* we assume now that u is provably non-negative *)
      (* note that this factorization is always possible, with a = 0 and b = v *)
      return mapping t ↦ { exact = u; rel = a; factor = u; abs = b }
  | otherwise: return nothing
  end case

type feto = None | Some (fe,proof_tree)

(* recursively generate a forward error formula and its proof *)
function generate(Δ: term ↦ fe, t:term) : feto
  (* FIXME: here we should ''lookup for terms equal to t of an interesting form''
     or, at least, if t is a defined constant, unfold it *)
  case on the form of t:
  |  t1 ⊕ t2 :
      return Some(generate_add(t,t1,t2,generate(Δ,t1),generate(Δ,t2))
  | otherwise:
     (* lookup table *)
     if Δ(t) = e then Some(e,Scall_prover PROVERS) else None
  end case
end function
```

Figure 1: Main structure of the algorithm for forward error generation.

Figure 1, explores recursively the given float term t, and tries to infer a forward error formula. It also builds a proof tree for proving that formula, under the form of a tree data structure as introduced in Section 2.2. In this tree, PROVERS denotes a list of provers, which is somehow arbitrary. In our examples we use the list [Alt-Ergo 2.5.4, CVC4 1.8, cvc5 1.0, Z3 4.12], which is enough, even with a short time limit of 1 second.

The first phase corresponds to the function `scan_hypot` of Figure 1. For each hypothesis of the appropriate forward error form, an entry in the table is recorded. The appropriate forms are two-fold: the

```
function generate_add(t:term,t1:term,t2:term,e1:fe,e2:fe): (fe,tree)
  case on the form of e1,e2:
  | None, None:
      (* we have no known errors on arguments, we just apply Property 3 *)
      let u = to_real t1 + to_real t2 in
      let e = { exact = u ; rel = ε; factor = |u|; abs = 0 } in
      Some (e, Scall_prover PROVERS)
  | Some (e1,T1), Some (e2,T2):
      (* we apply Lemma 3.1 *)
      let e = { exact = e1.exact + e2.exact; rel = ε + e1.rel+e2.rel;
                factor = e1.factor+e2.factor;
                abs = e1.abs(1+ε+e2.rel)+e2.abs(1+ε+e1.rel) }
      in
      let φ = |t−e.exact| ≤ e.rel×e.factor+e.abs in
      let T =
        Sapply_trans("assert", [φ], [
          Sapply_trans("apply", ["add_propagation"], [T1; T2; Scall_prover PROVERS; ...])])
      in
      Some (e, T)
  | Some (e1,T1), None:
      (* we apply Lemma 3.1 with no errors on second argument *)
      let u2 = to_real t2 in
      let e = { exact = e1.exact+u2; rel = ε+e1.rel+0;
                factor = e1.factor+|u2|;
                abs = e1.abs×(1+ε+0)+0 }
      in
      let φ = |t−e.exact| ≤ e.rel×e.factor+e.abs in
      let T =
        Sapply_trans("assert", [φ], [
          Sapply_trans("apply", ["add_propagation"], [T1; Scall_prover PROVERS; ...])])
      in
      Some (e,T)
  | None, Some(e2,T2):
      (* symmetrical case as the one above *)
      ...
```

Figure 2: Inference of forward error formulas for addition.

first form is

$$|t-u| \leq a \times |u| + b$$

and the second is similar

$$|t-u| \leq a \times u + b$$

without the absolute value on the right. This second form may indeed occur when it is known that the term u is non-negative. In each case, u is taken as the exact value, a the relative error, b the constant error, and $|u|$ (resp. u) the factor.

The second phase examines the shape of the given term t. As shown on Figure 1, the only case we

```
let addition_errors_basic (a b c : usingle)
  ensures {
    let exact = to_real a +. to_real b +. to_real c in
    let exact_abs = abs (to_real a) +. abs (to_real b) +. abs (to_real c) in
    abs (to_real result -. exact) <=. 2. *. eps *. exact_abs
  }
= a ++. b ++. c
```

Figure 3: A toy program computing the sum of three numbers.

consider for now is when it is a floating-point addition. The case distinction there is the one that will be extended in remaining sections. When t has the form $t_1 \oplus t_2$ the algorithm recursively generate forward errors for t_1 and t_2, and then the extra function $\texttt{generate}_{add}$ is called to combine them. The pseudo-code for the latter function is shown on Figure 2. There are four cases depending whether a forward error for t_1 (resp. t_2) was inferred or not. The first case is when none of t_1 and t_2 have a forward error inferred. In that case, we simply invoke Property 3 and provide a forward error w.r.t. the real addition $t_1 + t_2$. The second case shown on Figure 2 is when both t_1 and t_2 have a forward error inferred. In that case we invoke Lemma 3.1. The third case is when t_1 has a forward error but not t_2, in that case we still invoke Lemma 3.1 in a degenerated case where $\tilde{y} = y$ and $y_{rel} = y_{abs} = 0$. The fourth case, not shown is similar by exchanging the roles of t_1 and t_2.

Notice that if the term has not the shape $t_1 \oplus t_2$ we lookup the table Δ for existence of a forward error for t from the hypotheses. This will be the last resort case later on, when no other known shape will be found.

3.3 Examples

We now look at some examples to show how the strategy is performed on them step by step.

Example 3.2 (Addition of three numbers)**.** *Figure 3 shows an example WhyML program performing the addition of three single-precision unbounded floating-point numbers a, b and c. It is specified with a post-condition stating a relative error of* 2ε *and absolute error equal to* 0*, with the factor* $|a| + |b| + |c|$*. To detail the process applied by the strategy, let us consider the excerpt of the proof task shown on Figure 4. The strategy is applied to the term* `result` *as argument. It performs the following steps:*

- *First, the task hypotheses are scanned with* `scan_hypot` *in order to populate* Δ*. Since no hypothesis has the form* $|\texttt{to\_real}\ t - v| \leq u$*,* Δ *is empty after the scan.*
- *Then the main algorithm runs with* `generate(Δ, result)`*.* `result` *has the form* $t_1 \oplus t_2$ *with* $t_1 = a \oplus b$ *and* $t_2 = c$*, therefore the function* `generate_add` *is called as follows*

  ```
  generate_add(t, a ⊕ b, c, generate(Δ, a ⊕ b), generate(Δ, c))
  ```

  The recursive call `generate(Δ, a ⊕ b)` *returns* `Some(fe, Scall_prover PROVERS)` *where* `fe` *is*

  ```
  { exact = to_real a + to_real b ;
    rel = ε;
    factor = |to_real a + to_real b|;
    abs = 0
  }
  ```

```
uadd_single_error_propagation :
  forall x_uf:usingle, y_uf:usingle, x:real, x':real, x_rel_err:real,
   x_abs_err:real, y:real, y':real, y_rel_err:real, y_abs_err:real.
   abs (to_real x_uf -. x) <=. ((x_rel_err *. x') +. x_abs_err) →
   abs (to_real y_uf -. y) <=. ((y_rel_err *. y') +. y_abs_err) →
   abs x <=. x' →
   abs y <=. y' →
   0.0 <=. x_rel_err →
   0.0 <=. y_rel_err →
   0.0 <=. x_abs_err →
   0.0 <=. y_abs_err →
   abs (to_real (x_uf ++. y_uf) -. (x +. y))
   <=. ((((x_rel_err +. y_rel_err) +. eps) *. (x' +. y'))
        +. ((((1.0 +. eps) +. y_rel_err) *. x_abs_err)
            +. (((1.0 +. eps) +. x_rel_err) *. y_abs_err)))

[...]

constant a : usingle

constant b : usingle

constant c : usingle

constant result : usingle = (a ++. b) ++. c

goal addition_errors_basic'vc :
  abs (to_real result -. ((to_real a +. to_real b) +. to_real c))
  <=. ((2.0 *. eps)
       *. ((abs (to_real a) +. abs (to_real b)) +. abs (to_real c)))
```

Figure 4: Sum of three numbers, excerpt of the proof task.

---

and the recursive call `generate(Δ, c)` *returns* `None`. *We are therefore in the third case of* `generate_add`, *which returns* `Some(fe, [Scall_prover PROVERS ;...])` *where* `fe` *is*

```
{ exact = to_real a + to_real b + to_real c ;
  rel = ε + ε ;
  factor = |to_real a + to_real b| + |to_real c| ;
  abs = 0;
}
```

The strategy thus asserts the formula

$$|\texttt{result - (to\_real } a + \texttt{to\_real } b + \texttt{to\_real } c)| \leq (\varepsilon+\varepsilon)(|\texttt{to\_real } a + \texttt{to\_real } b| + |\texttt{to\_real } c|) + 0$$

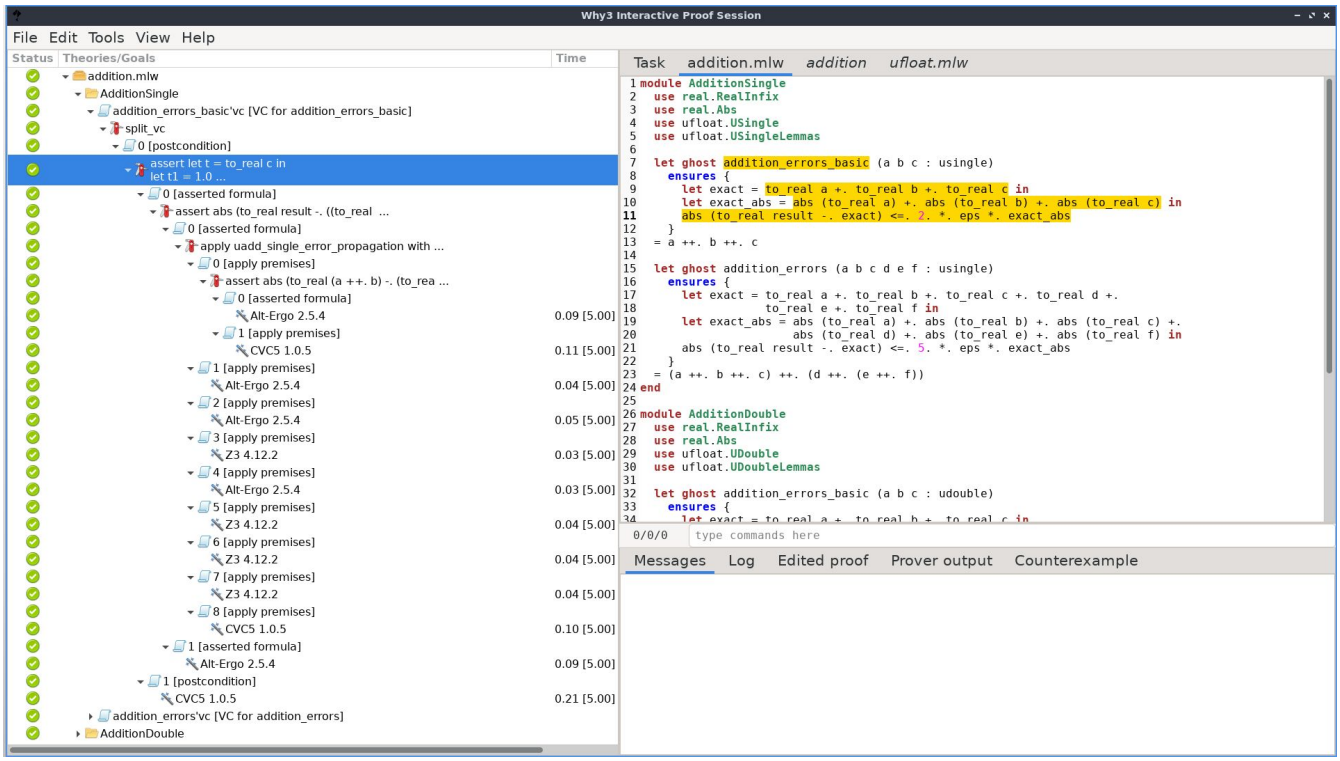


Figure 5: Detailed proof of the addition of three numbers, in the Why3 IDE. After selecting the post-condition goal in the tree view on the left, the strategy `forward_propagation result` was invoked. It automatically generated the proof tree rooted at the "assert" node highlighted in blue. All the subtasks generated were resolved by the provers that were tried in parallel (the quickest one being retained), among Alt-Ergo 2.5.4, cvc5 1.0.5 and Z3 4.12.2, in a fraction of a second.

---

together with an invocation of the `apply` *transformation with lemma* `uadd_single_error_propagation` *and appropriate arguments for its quantified variables. The generated subtasks are then proved automatically by our set of automated provers. Finally the given post-condition is automatically proved by automated provers using the asserted formula: they just have to apply the triangular inequality on* $|$`to_real` $a +$ `to_real` $b|$ *and simplify with algebraic laws of rings. Figure 5 shows how this proof is made in practise within Why3 IDE. The main lesson to draw is that the proof is made automatically, whereas, without the strategy, no prover was able to solve the initial goal, even with a large time limit.*

Example 3.3 (Addition of six numbers)**.** *The previous example can be generalised to addition of more numbers. For example, if given a code adding six numbers, the following post-condition can be automatically proved by the propagation strategy.*

$$|(a \oplus b \oplus c \oplus d \oplus e \oplus f) - (a+b+c+d+e+f)| \leq 5\varepsilon(|a|+|b|+|c|+|d|+|e|+|f|)$$

Example 3.4 (Addition with an uninterpreted function)**.** *It may be questionable whether the strategy works only when adding variables. Indeed no, the argument of addition can be somehow arbitrary. This example illustrates this case. Figure 6 shows a simple program that adds two single-precision numbers,*

```
function usqrt (x:usingle) : usingle

let ghost add_sqrt (a b : usingle)
  ensures {
    let exact = to_real a +. to_real (usqrt b) in
    abs (to_real result -. exact) <=. eps *. abs exact
  }
= a ++. usqrt b
```

Figure 6: Addition with an uninterpreted function.

one of which is the result of a function `usqrt` *with no specification, in other words* uninterpreted. *Here the strategy performs the following steps:*

- *First, the task hypotheses are scanned with* `scan_hypot` *in order to populate* Δ. *Since no hypothesis has the form* `|to_real t - v| <= u`, Δ *is empty after the scan.*
- *Then, the main algorithm proceeds with* `generate(`Δ`, t)`. `t` *has the form* $t_1 \oplus t_2$, *therefore the call*

  ```
  generate_add(t, a, usqrt(b), generate(Δ, a), generate(Δ, usqrt(b)))
  ```

  is invoked. Recursively, `generate(`Δ`, a)` *and* `generate(`Δ`, usqrt(b))` *both return* `None`. *We are therefore in the first case of* `generate_add`, *which returns* `Some(fe, [Scall_prover PROVERS])` *where* `fe` *is*

  ```
  { exact = to_real a + to_real (usqrt b) ;
    rel = ε;
    factor = |to_real a + to_real (usqrt b)| ;
    abs = 0;
  }
  ```

At the end the following forward error formulas is generated and proved automatically:

$$|\texttt{to\_real result} - (a + \texttt{usqrt } b)| \leq \varepsilon |a + \texttt{usqrt } b|$$

4 Subtraction and Multiplication

The extension of the strategy to subtraction and multiplication goes naturally by stating propagation lemmas for these two operations, and then extending the algorithm.

4.1 Propagation lemmas

The lemma for subtraction is essentially the same as the one for addition since both operations share properties 3, 4 and 5. Indeed the lemma for subtraction can be derived from the one of addition.

Lemma 4.1 (Error propagation through a subtraction)**.** *For any real numbers* $x, y, \tilde{x}, \tilde{y}, x_{rel}, y_{rel}, x_{abs}, y_{abs}$, *and for any floats* $\hat{x}$ *and* $\hat{y}$ *(supposed to denote approximations of* x *and* y, *respectively), such that:*

- $|\hat{x} - x| \leq x_{rel}\tilde{x} + x_{abs}$

- $|\hat{y}-y| \leq y_{rel}\tilde{y}+y_{abs}$
- $|x| \leq \tilde{x}$
- $|y| \leq \tilde{y}$
- $x_{rel}, y_{rel}, x_{abs}, y_{abs} \geq 0$

we have

$$|(\hat{x} \ominus \hat{y}) - (x-y)| \leq (x_{rel}+y_{rel}+\varepsilon)(\tilde{x}+\tilde{y}) + x_{abs}(1+\varepsilon+y_{rel}) + y_{abs}(1+\varepsilon+x_{rel})$$

Proof. Apply Lemma 3.1 to the respective opposites of $\hat{y}$ and y, and use the fact taking the opposite is an exact floating-point operation[2]. □

We now devise a propagation lemma for multiplication, but first we need to state and prove an auxiliary lemma.

Lemma 4.2. *For any real numbers $x, \tilde{x}, z, x_{rel}, y_{rel}$, and for any float $\hat{x}$ (supposed to denote an approximation of x), such that:*

- $|\hat{x}-x| \leq x_{rel}\tilde{x}+x_{abs}$
- $|x| \leq \tilde{x}$
- $x_{rel}, x_{abs} \geq 0$

we have

$$|\hat{x}z - xz| \leq x_{rel}\tilde{x}|z| + x_{abs}|z|$$

Proof.

$$|\hat{x}z - xz| = |(\hat{x}-x)z| = |(\hat{x}-x)| \times |z| \leq (x_{rel}\tilde{x}+x_{abs})|z|$$

□

Lemma 4.3 (Error propagation through multiplication)**.** *For any real numbers $x, y, \tilde{x}, \tilde{y}, x_{rel}, y_{rel}, x_{abs}, y_{abs}$, and for any floats $\hat{x}$ and $\hat{y}$ (supposed to denote approximations of x and y, respectively), such that:*

- $|\hat{x}-x| \leq x_{rel}\tilde{x}+x_{abs}$
- $|\hat{y}-y| \leq y_{rel}\tilde{y}+y_{abs}$
- $|x| \leq \tilde{x}$
- $|y| \leq \tilde{y}$
- $x_{rel}, y_{rel}, x_{abs}, y_{abs} \geq 0$

we have

$$\begin{aligned} |(\hat{x} \otimes \hat{y}) - xy| \quad \leq \quad & (\varepsilon + (x_{rel}+y_{rel}+x_{rel}y_{rel})(1+\varepsilon))\tilde{x}\tilde{y} + \\ & (\tilde{x}(y_{abs}+x_{rel}y_{abs}) + \tilde{y}(x_{abs}+y_{rel}x_{abs}) + x_{abs}y_{abs})(1+\varepsilon) + \eta \end{aligned}$$

[2] that is, more precisely, `to_real` $(\ominus\hat{y}) = -$`to_real` $\hat{y}$

Proof. We have

$$\begin{aligned}
\hat{x}\hat{y} &\leq x\hat{y} + x_{rel}\tilde{x}|\hat{y}| + x_{abs}|\hat{y}| && \text{(by Lemma 4.2 with } z = \hat{y}\text{)}\\
&\leq xy + y_{rel}|x|\tilde{y} + y_{abs}|x| + x_{rel}\tilde{x}|\hat{y}| + x_{abs}|\hat{y}| && \text{(by Lemma 4.2 with } z = x\text{)}\\
&\leq xy + y_{rel}|x|\tilde{y} + y_{abs}|x| + x_{rel}\tilde{x}y + y_{rel}x_{rel}\tilde{x}\tilde{y} + x_{rel}\tilde{x}y_{abs} + x_{abs}|\hat{y}| && \text{(by Lemma 4.2 with } z = x_{rel}\tilde{x}\text{)}\\
&\leq xy + y_{rel}|x|\tilde{y} + y_{abs}|x| + x_{rel}\tilde{x}y + y_{rel}x_{rel}\tilde{x}\tilde{y} + x_{rel}\tilde{x}y_{abs} + x_{abs}y + y_{rel}x_{abs}\tilde{y} + x_{abs}y_{abs}\\
& && \text{(by Lemma 4.2 with } z = x_{abs}\text{)}\\
&\leq xy + y_{rel}\tilde{x}\tilde{y} + y_{abs}\tilde{x} + x_{rel}\tilde{x}\tilde{y} + y_{rel}x_{rel}\tilde{x}\tilde{y} + x_{rel}\tilde{x}y_{abs} + x_{abs}\tilde{y} + y_{rel}x_{abs}\tilde{y} + x_{abs}y_{abs}\\
&\leq xy + (x_{rel} + y_{rel} + x_{rel}y_{rel})\tilde{x}\tilde{y} + \tilde{x}(y_{abs} + x_{rel}y_{abs}) + \tilde{y}(x_{abs} + y_{rel}x_{abs}) + x_{abs}y_{abs}
\end{aligned}$$

Finally, we have

$$\begin{aligned}
\hat{x}\otimes\hat{y} - xy &= (\hat{x}\otimes\hat{y} - \hat{x}\hat{y}) + (\hat{x}\hat{y} - xy)\\
&\leq (\varepsilon|\hat{x}\hat{y}| + \eta) + (\hat{x}\hat{y} - xy) && \text{(by Property (2))}\\
&\leq (\varepsilon|\hat{x}\hat{y}| + \eta) + (x_{rel} + y_{rel} + x_{rel}y_{rel})\tilde{x}\tilde{y} + \tilde{x}(y_{abs} + x_{rel}y_{abs}) + \tilde{y}(x_{abs} + y_{rel}x_{abs}) + x_{abs}y_{abs}\\
&\leq (\varepsilon|xy + (x_{rel} + y_{rel} + x_{rel}y_{rel})\tilde{x}\tilde{y} + \tilde{x}(y_{abs} + x_{rel}y_{abs}) + \tilde{y}(x_{abs} + y_{rel}x_{abs}) + x_{abs}y_{abs}| + \eta)\\
&\quad + (x_{rel} + y_{rel} + x_{rel}y_{rel})\tilde{x}\tilde{y} + \tilde{x}(y_{abs} + x_{rel}y_{abs}) + \tilde{y}(x_{abs} + y_{rel}x_{abs}) + x_{abs}y_{abs}\\
&\leq (\varepsilon + (x_{rel} + y_{rel} + x_{rel}y_{rel})(1+\varepsilon))\tilde{x}\tilde{y}\\
&\quad + (\tilde{x}(y_{abs} + x_{rel}y_{abs}) + \tilde{y}(x_{abs} + y_{rel}x_{abs}) + x_{abs}y_{abs})(1+\varepsilon) + \eta
\end{aligned}$$

We proceed similarly to prove the lower bound for $\hat{x}\otimes\hat{y} - xy$. □

4.2 Increment in the Strategy

To support subtraction and multiplication, we add new cases in the `generate` function. We also define a $\texttt{generate}_{sub}$ function and a $\texttt{generate}_{mul}$ function. $\texttt{generate}_{sub}$ is the same as $\texttt{generate}_{add}$ but with subtraction instead of additions. $\texttt{generate}_{mul}$ is defined in Figure 7.

4.3 Examples

For the following examples, we won't detail the step by step process of the strategy as it should be pretty clear by now.

Example 4.4. *Error on $(a \oplus b \ominus c) \otimes d$. With the application of Property 3 in $\texttt{generate}_{add}$ and the application of lemma 4.1 in $\texttt{generate}_{sub}$, the strategy computes*

$$|(a \oplus b) \ominus c - (a + b - c)| \leq 2\varepsilon|a + b| + |c|$$

Then by applying Lemma 4.3 in $\texttt{generate}_{mul}$ the strategy finds the final error bound

$$|((a \oplus b) \ominus c) \otimes d - (a + b - c)d| \leq 3\varepsilon(1+\varepsilon)(|a + b| + |c|)|d| + \eta$$

Example 4.5. *Determinant of a 2x2 matrix*

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

that is $a \otimes d \ominus b \otimes c$. First, by two applications of Property 2 in $\texttt{generate}_{mul}$, the strategy computes

$$|a \otimes d - ad| \leq \varepsilon|ad| + \eta$$

```
function generate_mul(t:term,t1:term,t2:term,e1:fe,e2:fe): (fe,tree)
  case on the form of e1,e2:
  | None, None:
      (* we have no known errors on arguments, we just apply Property 2 *)
      let u = to_real t1 × to_real t2 in
      let e = { exact = u ; rel = ε; factor = |u|; abs = η } in
      Some (e, Scall_prover PROVERS)
  | Some (e1,T1), Some (e2,T2):
      (* we apply Lemma 4.3 *)
      let e = { exact = e1.exact×e2.exact;
                rel = ε+(e1.rel+e2.rel+e1.rel×e2.rel)(1+ε);
                factor = e1.factor×e2.factor;
                abs = (e1.factor(e2.abs+e2.abs×e1.rel)+e1.factor(e2.abs+e2.abs×e1.rel)
                  +e1.abs×e2.abs)(1+ε)+η;
              }
      in
      let φ = |t−e.exact| ≤ e.rel×e.factor+e.abs in
      let T =
        Sapply_trans("assert", [φ], [
          Sapply_trans("apply", ["mul_propagation"], [T1; T2; Scall_prover PROVERS; ...])])
      in
      Some (e, T)
  | Some (e1,T1), None:
      (* we apply Lemma 4.3 with no errors on second argument *)
      let u2 = to_real t2 in
      let e = { exact = e1.exact×u2;
                rel = ε+(e1.rel+0)(1+ε);
                factor = e1.factor×|u2|;
                abs = (0+|u2|(e1.abs+0)+0)(1+ε)+η;
              }
      in
      let φ = |t−e.exact| ≤ e.rel×e.factor+e.abs in
      let T =
        Sapply_trans("assert", [φ], [
          Sapply_trans("apply", ["mul_propagation"], [T1; Scall_prover PROVERS; ...])])
      in
      Some (e,T)
  | None, Some(e2,T2):
      (* symmetrical case as the one above *)
      ...
```

Figure 7: Generation of forward error formulas through multiplication.

and

$$|b \otimes c - bc| \leq \varepsilon |bc| + \eta$$

Then by applying Lemma 4.1 (in $\texttt{generate}_{sub}$*) we get the following*

$$|(a \otimes d \ominus b \otimes c) - (ad - bc)| \leq 3\varepsilon(|ad| + |bc|) + 2\eta(1 + 2\varepsilon)$$

Example 4.6. *Square of the norm of a 3D-vector* $(a\ b\ c)$ *that is* $a \otimes a \oplus b \otimes b \oplus c \otimes c$*.*
By applying Property 2 three times, then Lemma 3.1 two times, the strategy computes

$$|a \otimes a \oplus b \otimes b \oplus c \otimes c - (a^2 + b^2 + c^2)| \leq 5\varepsilon(a^2 + b^2 + c^2) + \eta(2(1 + 2\varepsilon)^2 + (1 + 4\varepsilon))$$

All the examples above are implemented in the Why3 set of examples, and proofs are done automatically after the strategy is invoked.

5 Support for Sine and Cosine Functions

The goal is now to extend the strategy to additional floating-point operations, namely the classical elementary functions such as sine, cosine, exponential, logarithm, etc. In this section, we explore how the support for such functions is added to the strategy, using the sine function as an example, and also the cosine function which is handled exactly the same way.

Some aspect that is significantly different from the handling of addition and other operations is that elementary functions are not normalised in the IEEE-754 standard, and generally speaking it might be quite demanding to assume implementations of approximations of such function with the best possible precision, on all the domain of floating-numbers. We want to proceed differently, that is to assume some implementation with a given precision, but not necessarily optimal, and possibly within a restricted domain of inputs. For this reason, we are not going to suppose the existence of a fixed implementation. Instead, the propagation lemmas are going to be expressed for one given value for which we compute an approximation, with an assumed precision. Such lemmas will be invoked for each particular value desired, in particular if such a value comes from some implementation.

5.1 Propagation Lemmas

The propagation lemma for the sine function is expressed as follows.

Lemma 5.1 (Error propagation through the sine function)**.** *For any real numbers* $x, \tilde{x}, x_{rel}, s_{rel}, x_{abs}, s_{abs}$*, and for any float* $\hat{x}$ *(supposed to denote an approximation of* x*) and* $\hat{s}$ *(supposed to denote an approximation of* $\sin(x)$*), such that.*

- $|\texttt{to\_real}\ \hat{x} - x| \leq x_{rel}\tilde{x} + x_{abs}$
- $|\texttt{to\_real}\ \hat{s} - \sin(\texttt{to\_real}\ \hat{x})| \leq s_{rel}|\sin(\texttt{to\_real}\ \hat{x})| + s_{abs}$
- $|x| \leq \tilde{x}$
- $0 \leq s_{rel}$

we have

$$|\texttt{to\_real}\ \hat{s} - \sin(x)| \leq |\sin(x)|s_{rel} + (x_{rel}\tilde{x} + x_{abs})(1 + s_{rel}) + s_{abs}$$

Proof. It is a known mathematical fact that for any a and b,

$$|\sin(a) - \sin(b)| \leq |a - b|$$

Therefore, we have

$$|\sin(\hat{x}) - \sin(x)| \leq |\hat{x} - x| \leq x_{rel}\tilde{x} + x_{abs}$$

```
(* generate function with sin support *)
function generate(Δ: term ↦ fe, t:term) : feto
  case on the form of t:
  |  t1 ⊕ t2 :
      return Some(generate_add(t,t1,t2,generate(Δ,t1),generate(Δ,t2))
    ...
  | otherwise:
    (* lookup table *)
    if Δ(t) = e then
      case on the form of e.exact
      | sin(to_real u) :
        case generate(Δ,u) of
        | None -> Some(e, Scall_prover PROVERS)
        | Some(e_u, T) -> Some(generate_sin(t, e, e_u, T))
      | otherwise: Some(e, Scall_prover PROVERS)
    else
      None
  end case
end function
```

Figure 8: `generate` function extended with support for sine.

Taking the upper bound, we have

$$\begin{aligned}
\hat{s} &\leq \sin(\hat{x}) + s_{rel}\,|\sin(\hat{x})| + s_{abs} \\
&\leq \sin(x) + x_{rel}\tilde{x} + x_{abs} + s_{rel}\,|\sin(x) + x_{rel}\tilde{x} + x_{abs}| + s_{abs} \\
&\leq \sin(x) + x_{rel}\tilde{x} + x_{abs} + s_{rel}\,|\sin(x)| + s_{rel}(x_{rel}\tilde{x} + x_{abs}) + s_{abs} \\
&= \sin(x) + (1 + s_{rel})(x_{rel}\tilde{x} + x_{abs}) + s_{rel}\,|\sin(x)| + s_{abs}
\end{aligned}$$

The proof is the similar for the lower bound. □

We have a very similar lemma for cosine, as follows, with a very similar proof.

Lemma 5.2 (Error propagation through the cosine function)**.** *For any real numbers $x, \tilde{x}, x_{rel}, c_{rel}, x_{abs}, c_{abs}$, and for any floats $\hat{x}$ and $\hat{c}$, such that*

- $|\texttt{to\_real}\ \hat{x} - x| \leq x_{rel}\tilde{x} + x_{abs}$
- $|\texttt{to\_real}\ \hat{c} - \cos(\texttt{to\_real}\ \hat{x})| \leq c_{rel}\,|\cos(\texttt{to\_real}\ \hat{x})| + c_{abs}$
- $|x| \leq \tilde{x}$
- $0 \leq c_{rel}$

we have

$$|\texttt{to\_real}\ \hat{c} - \cos(x)| \leq |\cos(x)|c_{rel} + (x_{rel}\tilde{x} + x_{abs})(1 + c_{rel}) + c_{abs}$$

```
(* Propagation of sin error *)
function generate_sin(t:term, e_s:fe, e_u:fe, T:proof_tree) : (fe * proof_tree)
  let e = { exact = sin(e_u.exact);
            rel = e_s.rel;
            factor = |sin(e_u.exact)|;
            abs = (e_u.rel e_u.factor + e_u.abs)(1 + e_s.rel) + e_s.abs
          }
  in
  let φ = |t − e.exact| ≤ e.rel e.factor + e.abs in
  let T =
    Sapply_trans("assert", [φ],
      Sapply_trans("apply", ["sin_propagation"], [T; Scall_prover PROVERS; ...]))
  in
  (e, T)
end function
```

Figure 9: Sine support in `generate` function

5.2 Increment in the Strategy Algorithm

For the strategy to support sine error propagation, we modify the function `generate` as shown in Figure 5.2 and we create a function $\texttt{generate}_{sin}$. The function `generate` is modified for the `otherwise` case. If e.exact has the form $\sin(\texttt{to\_real}\ u)$ for some float expression u, then we consider that t is some approximation of $\sin(u)$. We thus query for some forward error for u. If we find some, we call the function $\texttt{generate}_{sin}$ which computes the combined forward error as the one given by Lemma 5.1. In any other case, that is either e.exact does not have the expected form, or no forward error is found for u, we just return the mapping of t that was found in Δ, as it was done previously.

The function $\texttt{generate}_{sin}$ is shown on figure 5.2.

The increment for cosine support is exactly the same except that we look for e.exact to have the form $\cos(\texttt{to\_real}\ u)$ instead of $\sin(\texttt{to\_real}\ u)$. The function $\texttt{generate}_{cos}$ is very similar to $\texttt{generate}_{sin}$.

5.3 Basic Examples

Our examples rely on some implementations of sine and cosine. These implementations are not fixed but described by contracts they must satisfy. The definitions of these functions in WhyML are shown in Figure 10. Consider the contract of the implementation of sine, called `sin_approx`: it is parameterised by three constants: a relative error `sin_rel_err` (denoted $E^{\text{rel}}_{\text{sin}}$ in the text below), an absolute error `sin_abs_err` (denoted $E^{\text{abs}}_{\text{sin}}$ below) and a maximal range for inputs `sin_max` (denoted M_{sin} below). The precondition states that the argument must be smaller than M_{sin}, whereas the post-condition states a forward error for the result. A parametric implementation of cosine is declared similarly.

Example 5.3. *Our first example is the computation of the sine of a sum,* $\sin(a+b)$. *Figure 11 shows this example as a WhyML program. The error bound that is inferred by the strategy is the following*

$$|\widehat{\sin}(a \oplus b) - \sin(a+b)| \leq E^{rel}_{\text{sin}} |\sin(a+b)| + \varepsilon(1 + E^{rel}_{\text{sin}})|a+b| + E^{abs}_{\text{sin}}$$

This result was inserted by hand in the source file of Figure 11 as a post-condition. That post-condition is thus proved trivially from the assertion inferred by the strategy.

```
constant sin_rel_err:real
axiom sin_rel_err_range : 0.0 <=. sin_rel_err
constant sin_abs_err:real
axiom sin_abs_err_range : 0. <=. sin_abs_err
constant sin_max:real
axiom sin_max_range : 0.0 <=. sin_max
val function sin_approx (x:usingle) : usingle
  requires { abs (to_real x) <=. sin_max }
  ensures { abs (to_real result -. sin (to_real x)) <=.
              sin_rel_err *. abs (sin (to_real x)) +. sin_abs_err }

constant cos_rel_err:real
axiom cos_rel_err_range : 0. <=. cos_rel_err
constant cos_abs_err:real
axiom cos_abs_err_range : 0. <=. cos_abs_err
constant cos_max:real
axiom cos_max_range : 0.0 <=. cos_max
val function cos_approx (x:usingle) : usingle
  requires { abs (to_real x) <=. cos_max }
  ensures { abs (to_real result -. cos (to_real x)) <=.
              cos_rel_err *. abs (cos (to_real x))  +. cos_abs_err }
```

Figure 10: Declarations of sine and cosine approximations.

```
let sin_simple_example (a b :udouble)
  requires { abs (to_real a) <=. 0.25 *. sin_max }
  requires { abs (to_real b) <=. 0.25 *. sin_max }
  ensures {
    abs (to_real result -. sin(to_real a +. to_real b))
    <=. sin_rel_err *. abs (sin (to_real a +. to_real b))
     +. eps *. abs (to_real a +. to_real b) *. (1. +. sin_rel_err)
     +. sin_abs_err
  }
= sin_approx(a ++. b)
```

Figure 11: Example program: the sine of the sum of 2 numbers.

We give here a few details on how the strategy proceed to discover the bound. First of all, see an excerpt of the proof task in Figure 12. We apply the strategy with the term `result` *as argument. The following steps are performed :*

- *First, the task hypotheses are scanned with* `scan_hypot` *in order to populate* Δ*. The only hypothesis having the form* $|$`to_real` $t - v| \leq u$ *is the one named* `Ensures`*, coming from the post-condition of the implementation of sine given Figure 10. That is, when the function* `sin_approx` *is called in the body of the function* `sin_basic` *of Figure 11, the VC generator produces an hypothesis with the instance of the post-condition of* `sin_approx`*. Therefore we have* Δ *mapping the result variable to*

```
sin_double_error_propagation :
 forall sinx_f:udouble, x_f:udouble, x_exact:real, x_factor:real, sin_rel:real,
        sin_abs:real, x_rel:real, x_abs:real.
  abs (to_real x_f -. x_exact) <=. ((x_rel *. x_factor) +. x_abs) ->
  abs (to_real sinx_f -. sin (to_real x_f))
    <=. ((sin_rel *. abs (sin (to_real x_f))) +. sin_abs) ->
  x_exact <=. x_factor ->
  0.0 <=. sin_rel ->
  abs (to_real sinx_f -. sin x_exact)
    <=. ((sin_rel *. abs (sin x_exact))
         +. ((((x_rel *. x_factor) +. x_abs) *. (1.0 +. sin_rel)) +. sin_abs))

[...]

constant a : udouble

constant b : udouble

constant result : udouble

Ensures :
  abs (to_real result -. sin (to_real (a ++. b)))
  <=. ((sin_rel_err *. abs (sin (to_real (a ++. b)))) +. sin_cst_err)

goal sin_simple_example'vc :
  abs (to_real result -. sin (to_real a +. to_real b))
  <=. (((sin_rel_err *. abs (sin (to_real a +. to_real b)))
        +. ((eps *. abs (to_real a +. to_real b)) *. (1.0 +. sin_rel_err)))
       +. sin_cst_err)
```

Figure 12: Excerpt of the proof task produced from the post-condition of example of Figure 11.

---

the forward error e_s equal to

$$\begin{array}{l}\{\ \texttt{exact} = \sin(\texttt{to\_real}(a \oplus b));\\ \quad \texttt{rel} = E^{\text{rel}}_{\sin};\\ \quad \texttt{factor} = |\sin(\texttt{to\_real}(a \oplus b))|;\\ \quad \texttt{abs} = E^{\text{abs}}_{\sin};\\ \}\end{array}$$

- *Then the main algorithm runs with* `generate(`Δ`, result)`. `result` *does not have the form of a basic float operation, however it has a mapping* `Some(`e`,`T`)` *in* Δ *with e.exact having the form* $\sin($`to_real` $u)$*, with u being* $x \oplus y$*. The function* `generate` *is then called recursively on u to compute its forward error. Because u has the form* $x \oplus y$ *the* $\texttt{generate}_{add}$ *function is called and returns the basic forward error* e_u *below.*

$$\begin{array}{l}\{\ \texttt{exact} = \texttt{to\_real}\ a + \texttt{to\_real}\ b\ ;\\ \quad \texttt{rel} = \varepsilon;\end{array}$$

```
  factor = |to_real a + to_real b|;
  abs = 0 ;
}
```

- *The algorithm then proceeds by calling* `generate`$_{sin}$`(result,` e_s`,` e_u`,` T`)` *which returns*

```
{ exact = sin(to_real a + to_real b)} ;
  rel = E_sin^rel ;
  factor = |sin(to_real a + to_real b)| ;
  abs = (ε|to_real a + to_real b| + 0)(1 + E_sin^rel) + E_sin^abs ;
}
```

Finally the strategy inserts the assertion

$$|result - \sin(\texttt{to\_real}\ a + \texttt{to\_real}\ b)| \leq$$
$$E^{rel}_{\sin}|\sin(\texttt{to\_real}\ a + \texttt{to\_real}\ b)| + \varepsilon|\texttt{to\_real}\ a + \texttt{to\_real}\ b|(1 + E^{rel}_{\sin}) + E^{abs}_{\sin}$$

in the proof task, together with a proof tree of it. All leaves of this proof tree are proved automatically, and the post-condition itself is proved from that assertion.

Example 5.4. *Another short example is* $\widehat{\cos}(a \ominus b)$*. The inferred error bound is*

$$|\widehat{\cos}(a \ominus b) - \cos(a-b)| \leq E^{rel}_{\cos}|\cos(a-b)| + \varepsilon(1 + E^{rel}_{\cos})|a-b| + E^{abs}_{\cos}$$

Example 5.5. *Let's consider the computation* $\widehat{\cos}(a) \otimes \widehat{\cos}(a) \oplus \widehat{\sin}(a) \otimes \widehat{\sin}(a)$*, we should give 1 if there were no rounding error. The assertion automatically produced by the strategy is textually as follows.*

```
let t1 = to_real a1 in
let t2 = sin t1 in
let t3 = cos t1 in
let t4 = 1.0 +. eps in
let t5 = t2 *. t2 in
let t6 = t3 *. t3 in
let t7 =
  eps +. (((cos_rel_err +. cos_rel_err) +. (cos_rel_err *. cos_rel_err)) *. t4)
in
let t8 =
  eps +. (((sin_rel_err +. sin_rel_err) +. (sin_rel_err *. sin_rel_err)) *. t4)
in
let t9 = (t4 *. (cos_abs_err +. (cos_abs_err *. cos_rel_err))) *. abs t3 in
let t10 = (t4 *. (sin_abs_err +. (sin_abs_err *. sin_rel_err))) *. abs t2 in
abs (to_real result -. (t6 +. t5))
<=. ((((t7 +. t8) +. eps) *. (abs t6 +. abs t5))
     +. (((t4 +. t8)
          *. (((t9 +. t9) +. (t4 *. (cos_abs_err *. cos_abs_err))) +. eta))
         +. ((t4 +. t7)
             *. (((t10 +. t10) +. (t4 *. (sin_abs_err *. sin_abs_err)))
                 +. eta))))
```

Notice that the formula is made shorter by introducing let bindings for the sub-term that occur repeatedly. This small trick allows to produce more readable formulas, which can indeed by copy-pasted and put as

expected post-conditions. Indeed that copy-paste can be made with simplification on-the-fly. On that particular example we derive the following simplified post-condition.

$$|\widehat{\cos}(a) \otimes \widehat{\cos}(a) \oplus \widehat{\sin}(a) \otimes \widehat{\sin}(a) - 1| \leq C_{rel} + S_{rel} + \varepsilon + (t + S_{rel})(tE^{abs}_{\cos}(2(1+E^{rel}_{\cos})|\cos a| + E^{abs}_{\cos}) + \eta) + (t + C_{rel})(tE^{abs}_{\sin}(2(1+E^{rel}_{\sin})|\sin a| + E^{abs}_{\sin}) + \eta)$$

where

$$\begin{aligned} t &= 1+\varepsilon \\ C_{rel} &= \varepsilon + tE^{rel}_{\cos}(2+E^{rel}_{\cos}) \\ S_{rel} &= \varepsilon + tE^{rel}_{\sin}(2+E^{rel}_{\sin}) \end{aligned}$$

5.4 Two Case Studies

The two following programs are concrete case studies from usual benchmarks for evaluating accuracy of numerical programs. The first example, `kinematics`, comes from the AxBench [37] benchmark and has been used to support the approach of Darulova and Volkova [15] to provide sound approximations of programs calling mathematical functions in Daisy. The second example, `raytracer`, comes from the SPEC CPU 2017 benchmark [11]. It has been used to support the approach of Briggs and Panchekha [10] to automatically identify mathematical functions implementations to achieve a good trade-off between accuracy and performance in a program that call them. Both use cases are generic functions that could be used for various applications in industrial mechanical engineering.

In these examples, some error bounds were computed automatically by the strategy. The results have been then inserted automatically in the source by hand. During this, the resulting formulas have been simplified, *e.g.* addition to zero, or multiplication by one were discarded.

Example 5.6. *Our first case study is called "Kinematics". The objective is to discover and prove an accuracy property when computing*

$$\frac{1}{2}\sin\theta_1 + \frac{5}{2}\sin(\theta_1+\theta_2)$$

for arbitrary values of θ_1 and θ_2. The WhyML code is shown on Figure 13 (ignore the post-condition for the moment). The formula automatically generated from the strategy applied to the term `result` *is textually as follows.*

```
let t1 = to_real o in
let t2 = to_real o2 in
let t3 = to_real theta11 in
let t4 = 1.0 +. eps in
let t5 = t3 +. to_real theta2 in
let t6 = t1 *. sin t3 in
let t7 = eps +. (sin_rel_err *. t4) in
let t8 = t2 *. sin t5 in
let t9 = t4 +. t7 in
abs (to_real result -. (t6 +. t8))
<=. ((((t7 +. t7) +. eps) *. (abs t6 +. abs t8))
     +. ((t9 *. (((t4 *. sin_abs_err) *. abs t1) +. eta))
         +. (t9
             *. (((t4
```

```
  let kinematics (theta1 theta2:usingle)
  requires { abs (to_real theta1) <=. 0.25 *. sin_max }
  requires { abs (to_real theta2) <=. 0.25 *. sin_max }
  ensures {
    let theta1 = to_real theta1 in
    let theta2 = to_real theta2 in
    let t1 = 1.0 +. eps in
    let t2 = eps +. sin_rel_err *. t1 in
    abs (to_real result -.
         (0.5 *. sin theta1 +. 2.5 *. sin (theta1 +. theta2)))
    <=.
    (* Relative part of the error *)
    (2.0 *. t2 +. eps) *.
       (0.5 *. abs (sin theta1) +. 2.5 *. abs (sin (theta1 +. theta2)))
    (* Absolute part of the error *)
    +. (t1 +. t2) *.
         (t1 *. (0.5 *. sin_abs_err
                +. 2.5 *. (eps *. abs (theta1 +. theta2) *. (1.0 +. sin_rel_err)
                           +. sin_abs_err))
          +. 2.0 *. eta)
  }
= exact_cte 0.5 **. sin_approx (theta1)
  ++. exact_cte 2.5 **. sin_approx(theta1 ++. theta2)
```

Figure 13: Example program : Kinematics.

```
                 *. (((eps *. abs t5) *. (1.0 +. sin_rel_err))
                     +. sin_abs_err))
                *. abs t2)
               +. eta))))
```

Notice also the use of the procedure of sub-term abstraction already mentioned above, which reduces the size of formulas by introducing common sub-terms as temporary variables.

This formula can be made more readable and slightly more compact by a few rewriting done by hand, to obtain the bound given by

$$
\begin{aligned}
&|\texttt{to\_real result} - \left(\frac{1}{2}\sin\theta_1 + \frac{5}{2}\sin(\theta_1+\theta_2)\right) \leq \\
&(2t_2+\varepsilon)\left(\frac{1}{2}|\sin\theta_1| + \frac{5}{2}|sin(\theta_1+\theta_2)|\right) \\
&+(t_1+t_2)\left[t_1\left(\frac{1}{2}E_{\sin}^{abs} + \frac{5}{2}\left(\varepsilon|\theta_1+\theta_2|(1+E_{\sin}^{rel}) + E_{\sin}^{abs}\right)\right) + 2\eta\right]
\end{aligned}
$$

with

$$
\begin{aligned}
t_1 &= 1+\varepsilon \\
t_2 &= \varepsilon + E_{\sin}^{rel} t_1
\end{aligned}
$$

That rewritten formula was copy-pasted into the original code of Figure 13 as a post-condition, and that post-condition is automatically proved from the forward error formula.

Example 5.7. *Our second case study is called "Raytracer". It amounts to compute the accuracy of the expression*

$$n_x \cos\theta \cos\phi + n_y \sin\theta + n_z \cos\theta \sin\phi$$

The WhyML code is shown in Figure 14. The formula that have been automatically computed by the strategy is shown on Figure 15. Simplifying by hand that formula results in the following error bound, inserted as a post-condition in the code of Figure 14

$$\begin{aligned}
&|\texttt{to_real result} - (n_x \cos\theta \cos\phi + n_y \sin\theta + n_z \cos\theta \sin\phi)| \leq \\
&\quad (t_4 + t_3^s + \varepsilon)(|nx \cos\theta \cos\phi| + |n_y \sin\theta| + |n_z \cos\theta \sin\phi|) + \\
&\quad (t_1 + t_3^s)[(t_1 + t_2^s)(t_1(E_{\cos}^{abs}(1 + t_2^c)|n_x \cos\theta| + (t_1 E_{\cos}^{abs}|nx| + \eta)((1 + E_{\cos}^{rel})|\cos\phi| + E_{\cos}^{abs})) + \eta) + \\
&\qquad\quad (t_1 + t_3^c)(t_1 E_{\sin}^{abs}|n_y| + \eta)] + \\
&\quad (t_1 + t_4)[t_1(E_{\sin}^{abs}(1 + t_2^c)|n_z \cos\theta| + (t_1 E_{\cos}^{abs}|n_z| + \eta)((1 + E_{\sin}^{rel})|\sin\phi| + E_{\sin}^{abs})) + \eta]
\end{aligned}$$

where

$$\begin{aligned}
t_1 &= 1 + \varepsilon \\
t_2^c &= \varepsilon + t_1 E_{\cos}^{rel} \\
t_2^s &= \varepsilon + t_1 E_{\sin}^{rel} \\
t_3^c &= \varepsilon + t_1(t_2^c + E_{\cos}^{rel} + t_2^c E_{\cos}^{rel}) \\
t_3^s &= \varepsilon + t_1(t_2^c + E_{\sin}^{rel} + t_2^c E_{\sin}^{rel}) \\
t_4 &= t_3^c + t_2^s + \varepsilon
\end{aligned}$$

As a partial conclusion of this section, we emphasise how efficient is the strategy to generate accuracy formulas. Those would be enormously difficult to find by hand.

(to be continued on next page after the figures)

```
let raytracer (theta phi nx ny nz:udouble)
  requires { abs (to_real phi) <=. sin_max }
  requires { abs (to_real phi) <=. cos_max }
  requires { abs (to_real theta) <=. sin_max }
  requires { abs (to_real theta) <=. cos_max }
  ensures {
    let nz = to_real nz in
    let ny = to_real ny in
    let nx = to_real nx in
    let phi = to_real phi in
    let theta = to_real theta in
    let t1 = 1.0 +. eps in
    let t2c = eps +. cos_rel_err *. t1 in
    let t2s = eps +. sin_rel_err *. t1 in
    let t3c = eps +. (t2c +. cos_rel_err +. t2c *. cos_rel_err) *. t1 in
    let t3s = eps +. (t2c +. sin_rel_err +. t2c *. sin_rel_err) *. t1 in
    let t4 = t3c +. t2s +. eps in
    abs (to_real result -.
         (nx *. cos theta *. cos phi +. ny *. sin theta +. nz *. cos theta *. sin phi))
    <=.
        (* Relative part of the error *)
        (t4 +. t3s +. eps) *.
        (abs (nx *. cos theta *. cos phi) +.
                    abs (ny *. sin theta) +. abs (nz *. cos theta *. sin phi))
        (* Absolute part of the error *)
        +. (t1 +. t3s)
            *. ((t1 +. t2s)
                 *. (t1 *. (cos_abs_err *. (1.0 +. t2c) *. abs (nx *. cos theta)
                            +. (t1 *. cos_abs_err *. abs nx +. eta) *.
                               ((1.0 +. cos_rel_err) *. abs (cos phi) +. cos_abs_err))
                     +. eta)
                +. (t1 +. t3c) *. (t1 *. sin_abs_err *. abs ny +. eta))
         +. (t1 +. t4)
            *. (t1 *. (sin_abs_err *. (1.0 +. t2c) *. abs (nz *. cos theta)
                       +. (t1 *. cos_abs_err *. abs nz +. eta) *.
                          ((1.0 +. sin_rel_err) *. abs (sin phi) +. sin_abs_err))
                +. eta)
  }
=
  nx **. cos_approx theta **. cos_approx phi
  ++. ny **. sin_approx theta
  ++. nz **. cos_approx theta **. sin_approx phi
```

Figure 14: Example program: Raytracer.

```
let t1 = to_real nz in
let t2 = to_real ny in
let t3 = to_real nx in
let t4 = to_real phi in
let t5 = to_real theta in
let t6 = sin t4 in
let t7 = cos t4 in
let t8 = cos t5 in
let t9 = 1.0 +. eps in
let t10 = t9 *. cos_abs_err in
let t11 = t1 *. t8 in
let t12 = t2 *. sin t5 in
let t13 = t3 *. t8 in
let t14 = eps +. (cos_rel_err *. t9) in
let t15 = eps +. (sin_rel_err *. t9) in
let t16 = t11 *. t6 in
let t17 = t13 *. t7 in
let t18 = (t10 *. abs t1) +. eta in
let t19 = (t10 *. abs t3) +. eta in
let t20 = eps +. (((t14 +. cos_rel_err) +. (t14 *. cos_rel_err)) *. t9) in
let t21 = eps +. (((t14 +. sin_rel_err) +. (t14 *. sin_rel_err)) *. t9) in
let t22 = (t20 +. t15) +. eps in
abs (to_real result -. ((t17 +. t12) +. t16))
<=. ((((t22 +. t21) +. eps) *. ((abs t17 +. abs t12) +. abs t16))
     +. (((t9 +. t21)
          *. (((t9 +. t15)
               *. (((((t9 *. (cos_abs_err +. (cos_abs_err *. t14)))
                      *. abs t13)
                     +. ((t9 *. (t19 +. (t19 *. cos_rel_err))) *. abs t7))
                    +. (t9 *. (t19 *. cos_abs_err)))
                   +. eta))
              +. ((t9 +. t20) *. (((t9 *. sin_abs_err) *. abs t2) +. eta))))
         +. ((t9 +. t22)
             *. (((((t9 *. (sin_abs_err +. (sin_abs_err *. t14)))
                    *. abs t11)
                   +. ((t9 *. (t18 +. (t18 *. sin_rel_err))) *. abs t6))
                  +. (t9 *. (t18 *. sin_abs_err)))
                 +. eta))))
```

Figure 15: The error propagation formula generated for the Raytracer example.

6 Support for Logarithm and Exponential

This section is dedicated to extension of the strategy to the exponential and logarithm functions. We proceed similarly as for sine and cosine, by stating and proving propagation lemmas of the appropriate shape, and then extend the core of the algorithm.

6.1 Propagation Lemmas

Lemma 6.1 (Error propagation through exponential)**.** *For any real numbers $x, \tilde{x}, x_{rel}, e_{rel}, x_{abs}, e_{abs}$, and any floats $\hat{x}$ (supposed to denote an approximation of x) and $\hat{e}$ (supposed to denote an approximation of $\exp(x)$), such that*

- $|\texttt{to\_real}\ \hat{x} - x| \leq x_{rel}\tilde{x} + x_{abs}$
- $|\texttt{to\_real}\ \hat{e} - \exp(\texttt{to\_real}\ \hat{x})| \leq e_{rel}\exp(\texttt{to\_real}\ \hat{x}) + e_{abs}$
- $x_{rel}, e_{rel}, x_{abs} \geq 0$
- $|x| \leq \tilde{x}$

we have

$$|\hat{e} - \exp(x)| \leq \exp(x)(e_{rel} + (\exp(x_{rel}\tilde{x} + x_{abs}) - 1)(1 + e_{rel})) + e_{abs}$$

Proof. We first show the formula

$$|\exp(\hat{x}) - \exp(x)| \leq \exp(x)(\exp(x_{rel}\tilde{x} + x_{abs}) - 1) \tag{7}$$

by distinguishing two cases for the upper bound and the lower bound. For the upper bound, by assumption we have

$$\hat{x} \leq x + x_{rel}\tilde{x} + x_{abs}$$

hence, since exp is monotonic,

$$\exp(\hat{x}) \leq \exp(x + x_{rel}\tilde{x} + x_{abs}) = \exp(x)\exp(x_{rel}\tilde{x} + x_{abs})$$

which entails Formula 7. Similarly, for the lower bound, by assumption we have

$$\hat{x} \geq x - x_{rel}\tilde{x} - x_{abs}$$

therefore

$$\exp(\hat{x}) \geq \exp(x - x_{rel}\tilde{x} - x_{abs}) = \exp(x)\exp(-x_{rel}\tilde{x} - x_{abs})$$

thus

$$\exp(\hat{x}) - \exp(x) \geq \exp(x)(\exp(-x_{rel}\tilde{x} - x_{abs}) - 1) \tag{8}$$

Furthermore, it is a known mathematical fact[3] that for any y, $\exp(y) + \exp(-y) \geq 2$, therefore

$$\exp(x_{rel}\tilde{x} + x_{abs}) + \exp(-x_{rel}\tilde{x} - x_{abs}) \geq 2$$

hence

$$\exp(-x_{rel}\tilde{x} - x_{abs}) - 1 \geq -\exp(x_{rel}\tilde{x} + x_{abs}) + 1$$

so

$$\exp(x)(\exp(-x_{rel}\tilde{x} - x_{abs}) - 1) \geq \exp(x)(-\exp(x_{rel}\tilde{x} + x_{abs}) + 1) = -\exp(x)(\exp(x_{rel}\tilde{x} + x_{abs}) - 1)$$

[3] $\exp(y) - 2 + \exp(-y) = \exp(y)(1 - 2\exp(-y) + (\exp(-y))^2) = \exp(y)(1 - \exp(-y))^2 \geq 0$

We have thus proved Formula 7.

To obtain the desired formula we again reason with two cases for the upper bound and the lower bound. For the upper bound we have $\hat{e} \leq \exp(\hat{x})(1+e_{rel})+e_{abs}$ so from Formula 7

$$\hat{e} - \exp(x) \leq \exp(x)(\exp(x_{rel}\tilde{x}+x_{abs})(1+e_{rel})-1)+e_{abs}$$

and the factor of $\exp(x)$ is

$$\begin{aligned}\exp(x_{rel}\tilde{x}+x_{abs})(1+e_{rel})-1 &= (\exp(x_{rel}\tilde{x}+x_{abs})-1)(1+e_{rel})+(1+e_{rel})-1\\ &= e_{rel}+(\exp(x_{rel}\tilde{x}+x_{abs})-1)(1+e_{rel})\end{aligned}$$

Similarly, for the lower bound, we have $\hat{e} \geq \exp(\hat{x})(1-e_{rel})-e_{abs}$ so from Formula 7

$$\begin{aligned}\hat{e} - \exp(x) &\geq (\exp(x)-\exp(x)(\exp(x_{rel}\tilde{x}+x_{abs})-1))(1-e_{rel})-\exp(x)-e_{abs}\\ &= -\exp(x)(1+((\exp(x_{rel}\tilde{x}+x_{abs})-1)-1)(1-e_{rel}))-e_{abs}\end{aligned}$$

and the factor of $-\exp(x)$ is

$$\begin{aligned}1+((\exp(x_{rel}\tilde{x}+x_{abs})-1)-1)(1-e_{rel}) &= 1+(\exp(x_{rel}\tilde{x}+x_{abs})-1)(1-e_{rel})-(1-e_{rel})\\ &= e_{rel}+(\exp(x_{rel}\tilde{x}+x_{abs})-1)(1-e_{rel})\\ &\leq e_{rel}+(\exp(x_{rel}\tilde{x}+x_{abs})-1)(1+e_{rel})\end{aligned}$$

□

Lemma 6.2 (Error propagation through logarithm)**.** *For any real numbers $x,\tilde{x},x_{rel},l_{rel},x_{abs},l_{abs}$, and any floats $\hat{x}$ (supposed to denote an approximation of x) and $\hat{l}$ (supposed to denote an approximation of* $\log(x)$*), such that*

- $|\hat{x}-x| \leq x_{rel}\tilde{x}+x_{abs}$
- $|\hat{l}-\log(\hat{x})| \leq l_{rel}|\log(\hat{x})|+l_{abs}$
- $x_{rel}, l_{rel} \geq 0$
- $0 < x \leq \tilde{x}$
- $x_{rel}\tilde{x}+x_{abs} < x$

we have

$$|\hat{l}-\log(x)| \leq l_{rel}|\log(x)| - \log\left(1-\frac{x_{rel}\tilde{x}+x_{abs}}{x}\right)(1+l_{rel})+l_{abs}$$

Proof. We first show the formula

$$|\log\hat{x}-\log x| \leq -\log\left(1-\frac{x_{rel}\tilde{x}+x_{abs}}{x}\right) \tag{9}$$

Pose $E = \dfrac{x_{rel}\tilde{x}+x_{abs}}{x}$. By the lemma hypotheses we have $0 \leq E < 1$. From the first hypothesis $|\hat{x}-x| \leq x_{rel}\tilde{x}+x_{abs}$, we get

$$x-(x_{rel}\tilde{x}+x_{abs}) \leq \hat{x} \leq x+x_{rel}\tilde{x}+x_{abs}$$

hence

$$x(1-E) \leq \hat{x} \leq x(1+E)$$

and thus

$$\log(x(1-E)) = \log(x) + \log(1-E) \leq \log(\hat{x}) \leq \log(x(1+E)) = \log(x) + \log(1+E)$$

Since $0 \leq E < 1$, we have $\log(1+E) \leq -\log(1-E)$, therefore:

$$\log(x) + \log(1-E) \leq \log(\hat{x}) \leq \log(x) - \log(1-E)$$

which proves Property (9). Now by taking the upper bound of the $\hat{l}$ error formula (second hypothesis) we have

$$\begin{aligned}
\hat{l} - \log(x) &\leq l_{rel} |\log(\hat{x})| + |\log(\hat{x}) - \log(x)| + r_{abs} \\
&\leq l_{rel} |\log(x)| + (1 + l_{rel}) |\log(\hat{x}) - \log(x)| + r_{abs} \\
&\leq l_{rel} |\log(x)| - (1 + l_{rel}) \log\left(1 - \frac{x_{rel}\tilde{x} + x_{abs}}{x}\right) + r_{abs} \qquad \text{(by (9))}
\end{aligned}$$

and similarly for the lower bound. □

6.2 Increment in the strategy

Adding the support for exponential and logarithm in our strategy is very similar to the support for sine and cosine. They are added through extra function $\texttt{generate}_{exp}$ and $\texttt{generate}_{log}$ as shown in Figure 6.2.

6.3 Examples

As we did with sin and cos, we can introduce support for log and exp function by assuming implementations of $\widehat{\log}$ and $\widehat{\exp}$ functions such that for any float t: if $0 < t \leq M_{\log}$ then

$$|\texttt{to\_real } \widehat{\log}(t) - \log(\texttt{to\_real } t)| \leq E^{\text{rel}}_{\log} |\log(\texttt{to\_real } t)| + E^{\text{abs}}_{\log}$$

and if $|t| \leq M_{\exp}$ then

$$|\texttt{to\_real } \widehat{\exp}(t) - \exp(\texttt{to\_real } t)| \leq E^{\text{rel}}_{\exp} |\exp(\texttt{to\_real } t)| + E^{\text{abs}}_{\exp}$$

where $M_{\log}$, $E^{\text{rel}}_{\log}$, $E^{\text{abs}}_{\log}$, $M_{\exp}$, $E^{\text{rel}}_{\exp}$ and $E^{\text{abs}}_{\exp}$ are some non-negative parameters.

For the following examples, we assume that we don't have an absolute error on $\widehat{\exp}$, eg. $E^{\text{abs}}_{\exp} = 0$. We also make the assumption that $E^{\text{rel}}_{\exp} \leq \frac{1}{2}$: this implies in particular that $\widehat{\exp}(x)$ is always positive, which is an property we need.

Example 6.3. *A first simple example is the accuracy of the computation for $\widehat{\log}(x \oplus y)$, which is inferred as such:*

$$|\widehat{\log}(x \oplus y) - \log(x+y)| \leq E^{rel}_{\log} |\log(x+y)| + (-\log(1-\varepsilon))(1 + E^{rel}_{\log}) + E^{abs}_{\log}$$

Example 6.4. *Another short example is $\widehat{\log}(\widehat{\exp}(x)) \oplus \widehat{\log}(\widehat{\exp}(y))$. Since the exact mathematical result is $x+y$, we can try to generated an accuracy formulas relating the result to $x+y$. The automatically inferred accuracy is*

$$\begin{aligned}
&|\widehat{\log}(\widehat{\exp}(x)) \oplus \widehat{\log}(\widehat{\exp}(y)) - (\log(\exp(x)) + \log(\exp(y)))| \\
&\quad \leq (2E^{rel}_{\log} + \varepsilon)(|\log(\exp(x))| + |\log(\exp(y))|) \\
&\quad +2(1 + E^{rel}_{\log} + \varepsilon)(-\log(1 - E^{rel}_{\exp})(1 + E^{rel}_{\log}) + E^{abs}_{\log})
\end{aligned}$$

```
(* Propagation of exp error *)
function generate_exp(t:term, e:fe, e':fe, T:proof_tree) : (fe * proof_tree)
  let e = { exact = exp(e'.exact);
            rel = e.rel+(exp(e'.rel×e'.factor+e'.abs)−1)(1+e.rel);
            factor = |exp(e'.exact)|;
            abs = e.abs
          }
  in
  let φ = |t−e.exact| ≤ e.rel × e.factor + e.abs in
  let T =
    Sapply_trans("assert", [φ], [
      Sapply_trans("apply", ["exp_propagation"], [T; Scall_prover PROVERS; ...])])
  in
  (e, T)
end function

(* Propagation of log error *)
function generate_log(t:term, e:fe, e':fe, T:proof_tree) : (fe * proof_tree)
  let e = { exact = log(e'.exact);
            rel = e.rel;
            factor = |log(e.exact)|;
            abs = −log(1−(e'.rel×e'.factor+e'.abs)/e'.exact)(1+e.rel)+e.abs
          }
  in
  let φ = |t−e.exact| ≤ e.rel × e.factor + e.abs in
  let T =
    Sapply_trans("assert", [φ], [
      Sapply_trans("apply", ["log_propagation"], [T; Scall_prover PROVERS; ...])])
  in
  (e, T)
end function
```

Figure 16: Exp and log support in `generate` function

from which can derive

$$\begin{aligned}|\widehat{\log}(\widehat{\exp}(x)) \oplus \widehat{\log}(\widehat{\exp}(y)) - (x+y)| \quad \leq \quad & (2E^{rel}_{\log} + \varepsilon)(|x|+|y|) \\ & +2(1+E^{rel}_{\log}+\varepsilon)(-\log(1-E^{rel}_{\exp})(1+E^{rel}_{\log})+E^{abs}_{\log})\end{aligned}$$

Example 6.5. *A still simple but slightly more involved example is the log-sum-exp function, studied in detail in a previous work of ours [8]. In this example we consider the case of a vector of fixed length, here 4. It means we want to bound the error on the expression*

$$\widehat{\log}(\widehat{\exp}(a_1) \oplus \widehat{\exp}(a_2) \oplus \widehat{\exp}(a_3) \oplus \widehat{\exp}(a_4))$$

To succeed, we have to assume that $E^{rel}_{\exp} \leq \frac{1}{8}$. *The source code of that example is shown in Figure 17.*

```
let lse4 (x1 x2 x3 x4 : udouble)
  requires { exp_error <=. 0x1p-3 }
  ensures {
    let exact =
      log (exp (to_real x1) +. exp (to_real x2) +.
           exp (to_real x3) +. exp (to_real x4))
    in
    abs (to_real result -. exact)
    <=. abs exact *. log_error
        -. log (1.0 -. (4.0 *. exp_error +. 3.0 *. eps))
           *. (1.0 +. log_error)
        +. log_cst_error
        }
= log_approx (exp_approx(x1) ++. exp_approx(x2) ++. exp_approx(x3) ++. exp_approx(x4))
```

Figure 17: The Log-Sum-Exp of a vector of dimension 4.

Our method allows us to automatically generate and prove the following accuracy property.

$$\begin{aligned}&|\widehat{\log}(\widehat{\exp}(a_1)\oplus\widehat{\exp}(a_2)\oplus\widehat{\exp}(a_3)\oplus\widehat{\exp}(a_4))-\log(\exp(a_1)+\exp(a_2)+\exp(a_3)+\exp(a_4))|\\&\quad\leq E_{\log}^{rel}|\log(\exp(a_1)+\exp(a_2)+\exp(a_3)+\exp(a_4))|\\&\qquad+(-\log(1-(4E_{\exp}^{rel}+3\varepsilon)))(1+E_{\log}^{rel})+E_{\log}^{abs}\end{aligned}$$

It should be noted that we can augment the size of the vector a above, and obtain a similar formula, showing an obvious regular pattern, for example for size 7:

$$\begin{aligned}&|\widehat{\log}(\widehat{\exp}(a_1)\oplus\widehat{\exp}(a_2)\oplus\widehat{\exp}(a_3)\oplus\widehat{\exp}(a_4)\oplus\widehat{\exp}(a_5)\oplus\widehat{\exp}(a_6)\oplus\widehat{\exp}(a_7))\\&-\log(\exp(a_1)+\exp(a_2)+\exp(a_3)+\exp(a_4)+\exp(a_5)+\exp(a_6)+\exp(a_7))|\\&\quad\leq E_{\log}^{\mathrm{rel}}|\log(\exp(a_1)+\exp(a_2)+\exp(a_3)+\exp(a_4)+\exp(a_5)+\exp(a_6)+\exp(a_7))|\\&\qquad+(-\log(1-(7E_{\exp}^{\mathrm{rel}}+6\varepsilon)))(1+E_{\log}^{\mathrm{rel}})+E_{\log}^{\mathrm{abs}}\end{aligned}$$

We can go further but while augmenting the size, we have to assume an even smaller bound on $E_{\exp}^{\mathrm{rel}}$: assuming $E_{\exp}^{\mathrm{rel}}\leq\frac{1}{16}$ we have been able to prove similar formulas for size 8 to 10. However, even if the strategy behaves well for such larger cases, the sub-goals generated become harder and harder to prove, that is they require a longer time to be solved: for size 10 as above, one of the sub-goals is prover by Alt-Ergo 2.6.0 in around 7 seconds. This a sub-goal that amounts to prove, from the resulting formula after application of the lemma for propagation through logarithm, which contains a division (see Lemma 6.2) a formula that has no division anymore. This technical inefficiency should be investigated in the future.

7 Conclusions

We proposed a methodology to automatically generate accuracy properties for numerical programs involving floating-point computations. The methodology supports calls to implementations of elementary functions like sine, cosine, exponential and logarithm, and the generated accuracy formulas are parameterised by the assumed accuracy of implementations of these functions. The trust in the generated formulas is high, thanks to the production of a formal proof that is constructed at the same time the accuracy formula is generated. These generated proofs are validated by Why3 transformations and external theorem provers.

We emphasise the importance of the reduced trusted code base of our approach: the generation algorithm does not need to be trusted since its result is double-checked afterwards.

7.1 Related Work

For related work regarding formal verification of floating-point programs, we refer to the related work section of Bonnot *et al.* [7, Section 5] which contains a fairly detailed discussion. Regarding the specific topic of automatically generating accuracy properties, some closely related work are the following.

In 2007, Akbarpour and Paulson [1] proposed a method for proving inequalities over real numbers, by combining first-order resolution with a decision procedure for closed field. Their approach supports functions like exponential and logarithm. It is implemented in the Metitarski prover. Being able to prove such inequalities is indeed a kind of preliminary assumption for our goal of proving accuracy formulas. Yet, using a prover like Metitarski on the proof tasks we have, even the simplest ones we have, is unlikely to work because establishing the accuracy property from scratch requires to discover the proper instances of the propagation lemmas, which appears to be too difficult. Our method is indeed a domain-specific proof tactic, that acts as a preliminary step before calling provers to solve required inequalities. In our experiments we use Alt-Ergo, cvc5 and Z3 for solving those inequalities, but Metitarski could be tried as well (it is a back-end solver supported by Why3). Notice that Metitarski is in fact using Z3 internally. The Metitarski tool is still maintained, recently in 2022 Coward *et. al* [13] showed an application to hardware verification.

In 2009, Daumas and Melquiond [16] proposed a method to compute bounds of mathematical expressions involving the rounding operator. The bounds are expressed by intervals of floating-point numbers, so that the error propagation proceeds by computation on such intervals: the so-called *interval arithmetic*. This method is the basis of the Gappa tool that is still maintained nowadays. The trust in the bounds generated is high, because Gappa can produce proofs that can be double-checked by the Coq proof assistant. In comparison to ours, this method can only compute constant bounds, whereas our method compute formulas depending on parameters. Moreover, their method does not support elementary functions like sine, cosine, exponential and logarithm. Interval arithmetic is though a significantly efficient approach, that has been used by others. In 2015, Martin-Dorel and Melquiond [30] used a combination of interval arithmetic and Taylor expansion to produce a method that can compute numerical bounds on univariate expressions. It is implemented as a Coq tactic: Coq-Interval. This method supports elementary functions. As already said for Metitarski, we could use Coq-interval as well as a solver for inequalities generated by our strategy, but still, it will not be powerful enough to discover the instances of propagation lemmas to apply.

In 2013, Gao *et. al* [24] introduced the tool dReal. It is an SMT solver specialised on nonlinear formulas over the reals. It supports multivariate functions and elementary functions. It implements a procedure that proceeds by decomposition of the domain of variables intro small enough blocks so as to prove the given formulas on each blocks. Coq-interval implements a similar approach, though only on one variable at once. dReal provides proof certificates in order to augment trust in its results. Again, dReal is also available as a back-end solver in Why3, and can indeed solve the inequalities generated by our strategy, but again cannot solve the initial proof tasks before the strategy is applied.

In 2022, Rasheed and Konecný [35, 34] proposed an enhanced decomposition approach to solve inequalities, implemented in the tool LPPaver which somehow improves over dReal and Metitarski. They also propose another tool PropaFP which, as its name suggest, attempts to apply a form of error propagation regarding floating-point rounding. Similarly to us, they apply the PropaFP tool to verification of computer programs, in particular programs written in SPARK [31], a subset of Ada for which formal verification is possible via translation to Why3. Their approach was clearly an inspiration to us. We were able to go further in practice by a deeper integration to Why3. We believe that our approach is also more powerful, because our propagation lemmas are more general than theirs.

7.2 Future Work

Along the line of our previous work [7, 8], we want to improve the tooling around the methodology. In particular, we would like to facilitate as much as possible the verification of numerical programs directly on their source code, typically written in the C language. We have already experimented the integration of our approach into the J^3 deductive verification plugin of TIS-Analyzer, a platform for static analysis of C programs. Any other programming language can naturally be a target, as soon as a Why3 front-end exists for it, which is the case for Ada and Rust [18].

For the future, we have complex case studies in mind, in particular programs involving rotation matrices. In that context, an open question is whether we can do better that estimating accuracy component by component: in principle better accuracy properties should be able to be found by bounding approximations in terms of the norms of the involved vectors and matrices.

References

[1] Behzad Akbarpour and Lawrence C. Paulson. Extending a resolution prover for inequalities on elementary functions. In *14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, number 4790 in Lecture Notes in Computer Science, pages 47–61. Springer-Verlag, 2007.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310. URL: http://hal.inria.fr/hal-00790310.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.gitlabpages.inria.fr/toccata/gallery/fm2012comp.en.html. URL: http://hal.inria.fr/hal-00967132/en, doi:10.1007/s10009-014-0314-5.

[6] Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, and Jean-Michel Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023. URL: https://hal.science/hal-04095151, doi:10.1017/S0962492922000101.

[7] Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché, and Raphaël Rieu-Helft. Formally verified bounds on rounding errors in concrete implementations of logarithm-sum-exponential functions. Research Report 9531, Inria, 2023. URL: https://inria.hal.science/hal-04343157.

[8] Paul Bonnot, Benoît Boyer, Florian Faissole, Claude Marché, and Raphaël Rieu-Helft. Formally verified rounding errors of the logarithm-sum-exponential function. In *Formal Methods in Computer-Aided Design*. IEEE, 2024. URL: https://inria.hal.science/hal-04674600.

[9] Martin Brain, Vijay D'silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, October 2014.

[10] Ian Briggs and Pavel Panchekha. Choosing mathematical function implementations for speed and accuracy. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 522–535, 2022.

[11] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[12] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL: https://hal.inria.fr/hal-01960203.

[13] Samuel Coward, Lawrence Paulson, Theo Drane, and Emiliano Morini. Formal verification of transcendental fixed- and floating-point algorithms using an automatic theorem prover. *Formal Aspects of Computing*, 34(2), 2022. doi:10.1145/3543670.

[14] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. Cr-libm: a correctly rounded elementary function library. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, volume 5205, pages 458–464. SPIE, 2003.

[15] Eva Darulova and Anastasia Volkova. Sound approximation of programs with elementary functions. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 174–183. Springer, 2019.

[16] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010. URL: http://www.lri.fr/~melquion/doc/09-toms.pdf, doi:10.1145/1644001.1644003.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[18] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verication of Rust programs. In *International Conference on Formal Engineering Methods - ICFEM*, Lecture Notes in Computer Science, Madrid, Spain, 2022. Springer. URL: https://hal.inria.fr/hal-03737878.

[19] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer. URL: https://hal.inria.fr/inria-00270820v1, doi:10.1007/978-3-540-73368-3_21.

[20] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. URL: http://hal.inria.fr/hal-00789533.

[21] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes in Computer Science*, pages 122–142, Rhodes, Greece, October 2020. Springer. See also `https://usr.lmf.cnrs.fr/~jcf/isola-2020/`. URL: `https://hal.inria.fr/hal-02696246`.

[22] Clément Fumex, Claude Marché, and Yannick Moy. Automated verification of floating-point computations in Ada programs. Research Report RR-9060, Inria, April 2017. URL: `https://hal.inria.fr/hal-01511183`.

[23] Clément Fumex, Claude Marché, and Yannick Moy. Automating the verification of floating-point programs. In Andrei Paskevich and Thomas Wies, editors, *Verified Software: Theories, Tools, and Experiments. Revised Selected Papers Presented at the 9th International Conference VSTTE*, number 10712 in Lecture Notes in Computer Science, Heidelberg, Germany, December 2017. Springer. URL: `https://hal.inria.fr/hal-01534533/`.

[24] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *24th Int. Conf. on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013. `doi:10.1007/978-3-642-38574-2_14`.

[25] Quentin Garchery. A framework for proof-carrying logical transformations. In *Proof eXchange for Theorem Proving*, volume 336 of *Electronic Proceedings in Theoretical Computer Science*, pages 5–23, July 2021. URL: `https://hal.archives-ouvertes.fr/hal-03349223`, `doi:10.4204/EPTCS.336.2`.

[26] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. `doi:10.1137/1.9780898718027`.

[27] IEEE standard for floating-point arithmetic, 2008. `https://dx.doi.org/10.1109/IEEESTD.2008.4610935`. `doi:10.1109/IEEESTD.2008.4610935`.

[28] Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 87:803–819, 2018. URL: `https://hal.inria.fr/hal-00934443`, `doi:10.1090/mcom/3234`.

[29] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In Tiziana Margaria and Bernhard Steffen, editors, *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478, Corfu, Greece, October 2016. Springer. URL: `https://hal.inria.fr/hal-01344110`, `doi:10.1007/978-3-319-47166-2_32`.

[30] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 2016. URL: `https://hal.inria.fr/hal-01086460`, `doi:10.1007/s10817-015-9350-4`.

[31] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. `doi:10.1017/CBO9781139629294`.

[32] Guillaume Melquiond. *Formal Verification for Numerical Computations, and the Other Way Around*. Habilitation à diriger des recherches, Université Paris Sud, April 2019. URL: `https://tel.archives-ouvertes.fr/tel-02194683`.

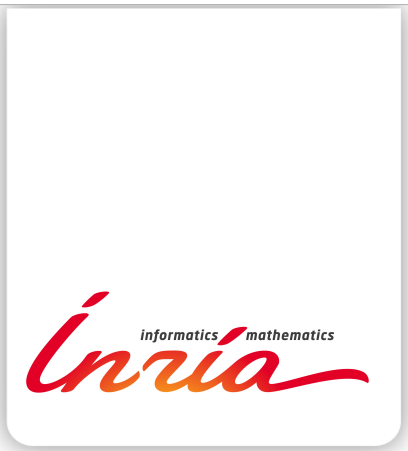
[33] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-point Arithmetic (2nd edition)*. Birkhäuser Basel, July 2018. URL: https://hal.inria.fr/hal-01766584, doi:10.1007/978-3-319-76526-6.

[34] Junaid Rasheed and Michal Konecný. Auto-active verification of floating-point programs via non-linear real provers. In Bernd-Holger Schlingloff and Ming Chai, editors, *20th Int. Conf. on Software Engineering and Formal Methods*, volume 13550 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2022. doi:10.1007/978-3-031-17108-6_2.

[35] Junaid Ali Rasheed. *Automatic Numerical Solving for Auto-active Verification of Floating-Point Programs*. PhD thesis, Aston University, 2022. URL: https://publications.aston.ac.uk/id/eprint/45270/.

[36] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH project. In *29th IEEE Symposium on Computer Arithmetic*, pages 26–34, 2022. URL: https://inria.hal.science/hal-03721525, doi:10.1109/ARITH54963.2022.00014.

[37] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34(2):60–68, 2016.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399