Deductive Formal Verification: How To Make Your Floating-Point Programs Behave

Sylvie Boldo

Habilitation à Diriger des Recherches

presentée à l'Université Paris-Sud

Spécialité: Informatique

par

Sylvie Boldo

Deductive Formal Verification: How To Make Your Floating-Point Programs Behave

Après avis de: Yves Bertot
John Harrison
Philippe Langlois

Soutenue le 6 octobre 2014 devant la commission d'examen formée de:

Marc Baboulin	Membre
Yves Bertot	Membre/Rapporteur
Éric Goubault	Membre
William Kahan	Membre
Philippe Langlois	Membre/Rapporteur
Claude Marché	Membre
Jean-Michel Muller	Membre

Thanks

First, I would like to thank my reviewers: Yves Bertot, John Harrison, and Philippe Langlois. Thanks for reading this manuscript and writing about it! They found bugs, typos, and English errors. Thanks for making this habilitation better.

Next of course are the jury members: Marc Baboulin, Yves Bertot, Éric Goubault, William Kahan, Philippe Langlois, Claude Marché, and Jean-Michel Muller. Thanks for coming! I am also thankful for the questions that were: *// to be ticked afterwards*

- ☐ numerous,
- ☐ interesting,
- ☐ broad,
- ☐ precise,
- ☐ unexpected,
- ☐ ..... *(please fill).*

Je passe maintenant au français pour remercier mon environnement habituel.

Un merci particulier à tous ceux qui m'ont poussée (avec plus ou moins de délicatesse) à écrire cette habilitation. La procrastination fut forte, mais vous m'avez aidée à la vaincre.

Pour l'environnement scientifique, merci aux membres des équipes ProVal puis Toccata et VALS. C'est toujours un plaisir d'avoir un support scientifique et gastronomique. C'est aussi grâce à vous que j'ai envie de venir travailler le matin.

Un merci spécial à mes divers co-auteurs sans qui une partie de ces travaux n'auraient juste pas eu lieu ou auraient été de notablement moindre qualité.

Un merci encore plus spécial à mes étudiantes en thèse: Thi Minh Tuyen Nguyen et Catherine Lelay. La thèse est une aventure scientifique et humaine. Merci de l'avoir partagée avec moi.

Mes profonds remerciements à deux extraordinaires assistantes: Régine Bricquet et Stéphanie Druetta. Mon travail n'est malheureusement pas que de la science, mais votre compétence m'a aidée à en faire plus.

Merci à mes enfants, Cyril et Cédric, de dormir (souvent) la nuit.

Merci Nicolas.

Contents

1. Introduction

If you compare French and English vocabulary, you will notice French only has the notion of "calcul", which ranges from kids' additions to Turing machines. In English, there is "calculation" for simple stuff and "computation" when it gets complicated. This manuscript is about floating-point *computations*. Roughly, floating-point arithmetic is the way numerical quantities are handled by the computer. It corresponds to the scientific notation with a limited number of digits for the significand and the exponent. Floating-point (FP) computations are in themselves complicated, as they have several features that make them non-intuitive. On the contrary, we may take advantage of them to create tricky algorithm which are faster or more accurate. In particular, this includes exact computations or obtaining correcting terms for the FP computations themselves. Finally, this covers the fact that, in spite of FP computations, we are able to get decent results, even if we sometimes have to pay for additional precision.

Why are we interested in FP computations? Because they are everywhere in our lives. They are used in control software, used to compute weather forecasts, and are a basic block of many hybrid systems: embedded systems mixing continuous, such as sensors results, and discrete, such as clock-constrained computations.

1.1 Back in Time

Let us go back (in time) to what computations are. More than just taxes and geometry, computations have long been used. A fascinating example is the Antikythera Mechanism, as it is a computing device both quite complex and very old.

The Antikythera Mechanism is a Greek device that dates from the 1st century BC [EF11, Edm13]. Found in 1900 by sponge divers, its remaining fragments were recently subjected to microfocus X-ray tomography and a bunch of software, including DNA matching. It was discovered its purpose was astronomical: it was a calendar with the position of the Sun and Moon, eclipse prediction, and probably the position of other planets and stars. This astronomical calculator was made of 30 toothed bronze gears wheels and several dials. It was a case approximately $34 \times 20 \times 10$ cm. Its upper-back dial was found to be a 235-month calendar, as 235 lunar months correspond almost exactly to 19 years. The Antikythera Mechanism was also able to predict possible lunar or solar eclipses, as observations show that eclipses happen 223 months later than another eclipse. A photo of a fragment and a replica are given in Figure 1.1 and a diagram showing the complexity is in Figure 1.2. Even a Lego replica has been set up[1].

The first mechanical computers such as the Antikythera Mechanism (80-60 BC), Pascal's calculator (1642), and Babbage's machines (1830) were all mechanical computers. Then floating-point appeared. The first electro-mechanical computers were those made by Konrad Zuse (1930s and 40s) including the famous Z3. Later came the well-known Crays (1970s). Now John and Jane Doe have a computer and a smartphone, able to perform more than 10^9

[1] `http://www.youtube.com/watch?v=RLPVCJjTNgk`

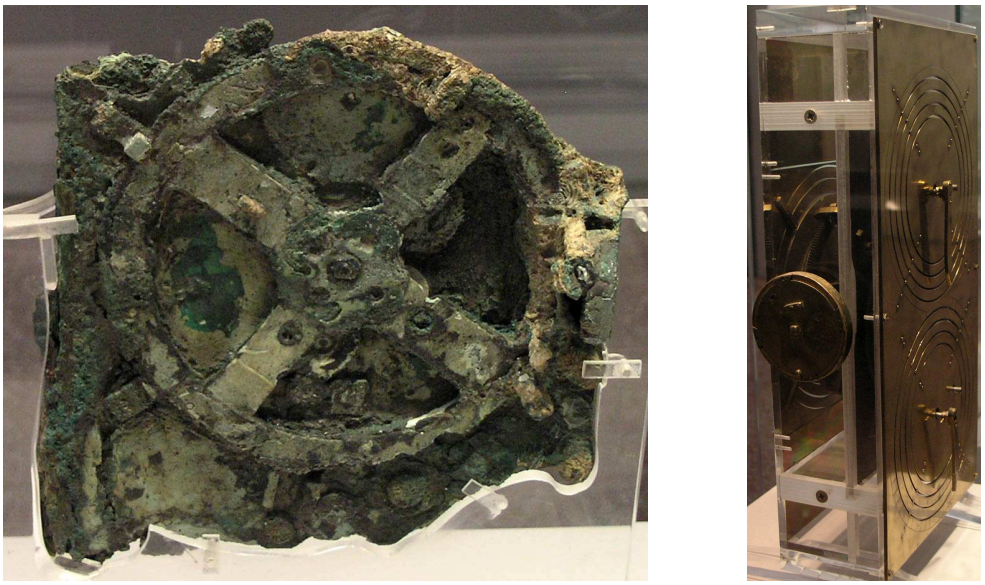

Figure 1.1: The Antikythera Mechanism: Photo of the A fragment on the left, Replica on the right. From `http://en.wikipedia.org/wiki/Antikythera_mechanism`.

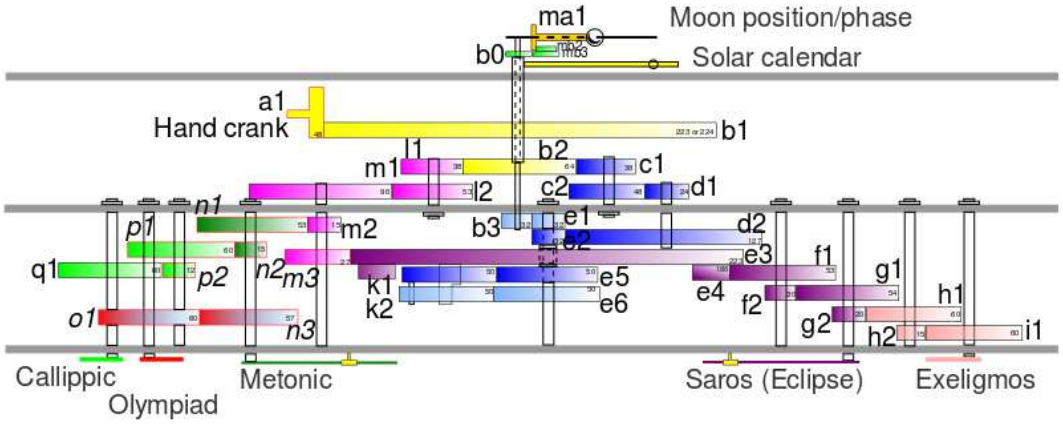

Figure 1.2: The Antikythera Mechanism: gear chain diagram for the known elements of the mechanism. Hypothetical gears in italics.
From `http://en.wikipedia.org/wiki/Antikythera_mechanism`.

FP computations per second. The (known) top supercomputer in November 2013[2] is Tianhe-2, developed by China's National University of Defense Technology, with a performance of 33×10^{15} FP computations per second, with more than 3,000,000 cores. Such a supercomputer can therefore compute more than 10^{23} bits of FP results in one day. Tucker [Tuc11] therefore has a valid question: "Are we just getting the wrong answers faster?".

[2] `http://www.top500.org`

1.2 Floating-Point Arithmetic

Let us dive into computer arithmetic. As in Konrad Zuse's Z3, nowadays computers use floating-point numbers. Which numbers and how operations behave on them is standardized in the IEEE-754 standard [IEE85] of 1985, which was revised in 2008 [IEE08].

We adopt here the level 3 vision of the standard: we do not consider bit strings, but the representation of floating-point data. The format will then be (β, p, e_{min}, e_{max}), where e_{min} and e_{max} are the minimal and maximal unbiased exponents, β is the radix (2 or 10), and p is the precision (the number of digits in the significand).

In that format, a floating-point number is then either a triple (s, e, m), or an exceptional value: $\pm\infty$ or a NaN (Not-a-Number). For non-exceptional values, meaning the triples, we have additional conditions: $e_{min} \leq e \leq e_{max}$ and the significand m has less than p digits. The triple can be seen as the real number with value

$$(-1)^s \times m \times \beta^e.$$

We will consider m as an integer and we therefore require that $m < \beta^p$. The other possibility is that m is a fixed-point number smaller than β. In this setting, the common IEEE-754 formats are `binary64`, which corresponds to (2, 53, −1074, 971) and `binary32`, which corresponds to (2, 24, −149, 104).

Infinities are signed and they have mathematical meaning (and will have in the computations too). The NaNs (Not-a-Numbers) are answers to impossible computations, such as 0/0 or $+\infty - +\infty$. They propagate in any computation involving a NaN. As infinities, they will be prevented in this manuscript: we will prove their absence instead of using or suffering from them.

Non-exceptional values give a discrete finite set of values, which can be represented on the real axis as in Figure 1.3. FP numbers having the same exponent are in a binade and are at equal distance from one to another. This distance is called the unit in the last place (ulp) as it is the intrinsic value of the last bit/digit of the significand of the FP number [MBdD+10]. When going from one binade to the next, the distance is multiplied by the radix, which gives this strange distribution. Around zero, we have the numbers having the smallest exponent and small mantissas, they are called subnormals and their ulp is that of the smallest normal number.

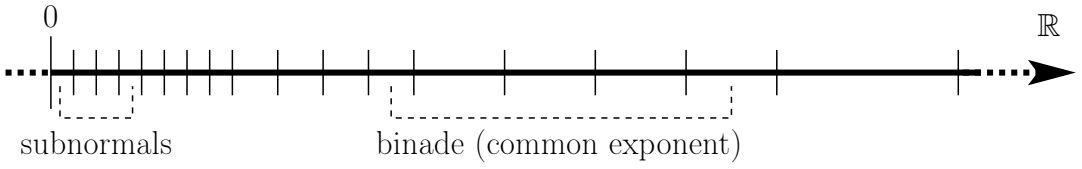

Figure 1.3: Distribution of the FP numbers over the real axis.

FP arithmetic tries to mimic real arithmetic but, in many cases, the exact result of an operation on two FP numbers is not a FP number (this will often be the case in this manuscript, but this is not representative). For example, in `binary64`, 1 and 2^{-53} are FP numbers, but $1 + 2^{-53}$ is not, as it would require 54 bits for the significand. The value therefore needs to be rounded. The IEEE-754 standard defines 5 rounding modes: towards positive, towards

negative, towards zero, rounding to nearest ties to even, and rounding to nearest ties away from zero.

The main rule of the IEEE standard of FP computation for basic operations is the following one, called *correct* rounding: each operation gives the same result as if it was first performed with infinite precision, and then rounded to the desired format. This is a very strong mathematical property that has several essential consequences:

- portability: as there is only one allowed result for an operation, you may change the processor, the programming language or the operating system without any change on the result. This is called recently "numerical reproducibility".
- accuracy: the error between the exact value and the computed value is small compared to these values. It is bounded by one ulp for directed rounding modes and half an ulp for roundings to nearest.

In particular, if we consider a format (β, p, e_{min}, e_{max}) with integer significand, then this error bound can be stated for any rounding to nearest $\circ$:

$$\forall x, \quad |x| \geq \beta^{p-1+e_{min}} \quad \Rightarrow \quad |\circ(x) - x| \leq \frac{\beta^{1-p}}{2} |\circ(x)|$$
$$\forall x, \quad |x| < \beta^{p-1+e_{min}} \quad \Rightarrow \quad |\circ(x) - x| \leq \frac{\beta^{e_{min}}}{2}$$

For directed rounding modes, the same formulas apply without the division by two. Correct rounding gives other interesting properties:

- It paves the way for mathematical proofs of what is happening in a computer program. A clear semantics is needed to build proofs that correspond to reality. A clean unique mathematical definition is therefore an advance towards safety.
- If the exact mathematical result of a computation fits in a FP format, then it is the returned result. This property will be thoroughly used in this manuscript. If we are able to prove that the result of a computation can be represented as a FP number, that is to say it fits in the destination format, then there will be no rounding, which is quite useful for the rest of the proof (such as bounding rounding error or proving algebraic equality).
- It helps to preserve some mathematical properties. For example, as both the square root and the rounding are increasing, the rounded square root is also increasing. Another example is: if a function is both correctly rounded and periodic, then its rounded counterpart is also periodic.

These properties only apply if the basic functions/operators are correctly rounded. In practice, this is required for addition, subtraction, multiplication, division, and square root. For these operators, three additional digits are enough to guarantee this correct rounding for all rounding modes, the last one being a sticky bit [Gol91]. Another operator available on recent processors is the FMA (fused multiply-and-add): it computes $a \times x + b$ with only one final rounding instead of a rounded multiplication followed by a rounded addition. Correct rounding is recommended for a larger set of functions, including pow, exp, sin, and cos, but it is known to be much more difficult to ensure [DDDM01].

For some ugly details, as for the difference between signaling and quiet NaNs, the sign of $0 - 0$ or the value of $\left(\sqrt{-0}\right)$, we refer the reader directly to the standard [IEE08]. Other major references are an article by Goldberg [Gol91] and the Handbook of Floating-Point Arithmetic [MBdD+10].

Previously, I write that this manuscript was about floating-point computations. This is not true. This manuscript is about *trust*.

1.3 Formal Proof Assistants

If an article or someone tells you something is true (such as a theorem, an algorithm, a program), will you believe it? This question is more sociological than scientific [DLP77]. You may believe it for many different reasons: because you trust the author, because the result seems reasonable, because everybody believes it, because it looks like something you already believe, because you read the proof, because you tested the program, and so on. We all have a trust base, meaning a set of persons, theorems, algorithms, programs we believe in. But we all have a different trust base, so to convince as many people as possible, you have to reduce the trust base as much as possible.

Proofs are usually considered as an absolute matter, and mathematicians are those that can discover these absolute facts and show them. However, mathematicians do fail. An old book of 1935 makes an inventory of 500 errors sometimes made by famous mathematicians [Lec35]. Given the current publication rate, such an inventory would now take several bookshelves. As humans are fallible, how can we make proofs more reliable? A solution is to rely on a less-fallible device, the computer, to check the proofs. We also need to detail the proofs much more than what mathematicians usually do.

We will need a "language of rules, the kind of language even a thing as stupid as a computer can use" (Porter) [Mac04]. Formal logic studies the languages used to define abstract objects such as groups or real numbers and reason about them. Logical reasoning aims at checking every step of the proof. It then removes any unjustified assumption and ensures that only correct inferences are used. The reasoning steps that are applied to deduce from a property believed to be true a new property believed to be true are called an inference rule. They are usually handled at a syntactic level: only the form of the statements matters, their content does not. For instance, the *modus ponens* rule states that, if both properties "A" and "if A then B" hold, then property "B" holds too, whatever the meaning of A and B. To check these syntactic rules are correctly applied, a thing as stupid as a computer can be used. It will be less error-prone and faster than a human being, as long as the proof is given in a language understandable by the computer. The drawback is that all the proof steps will need to be supported by evidence. Depending on your trust base, there are two different approaches. The first one follows the LCF approach [Gor00], meaning the only thing to trust is a small kernel that entirely checks the proof. The rest of the system, such as proof search, may be bug-ridden as each proof step will ultimately be checked by the trusted kernel. In this case, the proof must be complete: there is no more skipping some part of the proof for shortening or legibility. Another approach is to trust the numerous existing tools in automated deduction, such as SMT solvers (see also Section 2.2).

Many formal languages and formal proof checkers exist. For the sociological and historical aspects of mechanized proofs, we refer the reader to MacKenzie [Mac04]. The description

and comparison of the most used ones would require another manuscript. Let me just cite a few sorted by alphabetical order; a comparison of those, focused on real analysis has been published [BLM14b].

The ACL2 system is based on a first-order logic with an inference rule for induction [KMM00]. It is widely and industrially used for processor verification (see below) as it is robust and has good prover heuristics. In ACL2, the user only submits definitions and theorems and the system tries to infer proofs. In case of proof failure, one can then add lemmas such as rewriting rules, give hints such as disabling a rule or instantiating a theorem. After developing a theory in ACL2, the user may execute it: for example after formalizing a microprocessor, it is possible to simulate a run of it.

I mainly worked with the Coq proof assistant [BC04, Coq14]. The Coq system is based on the Calculus of Inductive Constructions which combines both a higher-order logic and a richly-typed functional programming language. Programs can be extracted from proofs to external programming languages such as OCaml, Haskell, or Scheme. As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define new proof methods. Coq follows the LCF approach with a relatively small kernel that is responsible for mechanically checking all the proofs.

The HOL Light system is based on classical higher-order logic with axioms of infinity, extensionality, and choice in the form of Hilbert's ϵ operator [Har09]. It also follows the LCF approach with a small kernel written in OCaml. Basic inference rules may be composed and proof search methods are supported. The user may also program his own automatic procedures. HOL Light was almost entirely written by Harrison. However, it builds on earlier versions of HOL, notably the original work by Gordon and Melham [GM93] and the improved implementation by Slind.

HOL4 is the fourth version of HOL. It follows the LCF approach with a small kernel written in SML. It builds on HOL98 but also on HOL Light ideas and tools. The logic is also the same [HOL12] as HOL Light's. External programs such as SMT or BDD engines can be called using an oracle mechanism.

The Mizar system is based on classical logic and the Jaskowski system of natural deduction [Try93, NK09]. A proof is a declarative-style script that is checked by the system. The primary goal is that the proofs be close to the mathematical vernacular, so that mathematicians may easily use it. This makes proofs longer but somehow more readable. The Mizar Mathematical Library is huge: 9 400 definitions of mathematical concepts and more than 49 000 theorems, that have been accumulated since 1989. The logic is based on an extension of the Zermelo-Fraenkel set theory. All objects are sets, and specific ones can be called integers or reals. The drawback is that Mizar is a fixed system: it cannot be extended or programmed by the user. There is no computational power nor user automation.

The PVS system is based on classical higher-order logic [ORS92]. Contrarily to the LCF approach to provers, PVS has no small kernel, but a large monolithic system containing the checker and many methods of automation. It heavily uses predicate subtypes and dependent types [ROS98]; TCCs (type-correctness conditions) are proof obligations generated by the PVS typechecker, for example to prevent division by zero. PVS has efficient decision procedures, that benefit from the typechecking information.

They may seem academic tools but they are not toys anymore. People are verifying hardware design on a daily basis, and are formally verifying compilers (see Section 4.2) and operating systems [KEH+09] too. I only use the Coq proof assistant in this manuscript, but this is not an important point. What is important is that all formal proofs are available from

my web page[3], so that they can be re-run to increase your trust in my statements.

Finally, I have often been asked this question: "are you very sure?". This can be restated as what trust can you put in a formal system, or as who checks the checker [Pol98]. As far as I am concerned, I have no strong belief in the absolute certainty of formal systems, I have a more pragmatic point of view, influenced by Davis [Dav72] with this nice title: "Fidelity in Mathematical Discourse: Is One and One Really Two?". It describes "Platonic mathematics" and its beliefs. In particular, Platonic mathematics assume that symbol manipulation is perfect. Davis thinks this manipulation is in fact wrong with a given probability. This probability depends if the manipulation is done by a human ($\approx 10^{-3}$) or a machine ($\approx 10^{-10}$). A cosmic ray may indeed mislead a computer to accept an incorrect proof. The probability is tiny, but nonzero. My opinion is that the computer is used here to decrease the probability of failure, and hence increase the trust. Perfection is not achievable, we are therefore looking for a strong guarantee that can be achieved using formal proof assistants.

1.4 State of the Art

1.4.1 Formal Specification/Verification of Hardware

A first subject where formal proofs have been applied is the verification of hardware and, in particular, the verification of the Floating-Point Unit (FPU). This is a logical consequence of the Pentium bug [Coe95]: the poor publicity was very strong and all hardware makers wanted to make sure this would never happen again. Formal methods are also useful as the implementation of computations with subnormal numbers is difficult [SST05].

Even before, formal methods have been applied to the IEEE-754 standard, and firstly by specifications: what does this standard in English really mean? Barrett wrote a such a specification in Z [Bar89]. It was very low-level and included all the features. Later, Miner and Carreño specified the IEEE-754 and 854 standards in PVS and HOL [Car95, Min95, CM95]. To handle similarly radixes 2 and 10, the definitions are more generic and not as low-level. The drawback is that few proofs have been made in this setting, except by Miner and Leathrum [ML96], and this casts a doubt on the usability of the formalization.

Concerning proofs made by processor makers, AMD researchers, particularly Russinoff, have proved in ACL2 many FP operators mainly for the AMD-K5 and AMD-K7 chips [MLK98, Rus98, Rus99, Rus00]. This is an impressive amount of proofs. The square root of the IBM Power4 chip [SG02] and a recent IBM multiplier have been verified using ACL2 too [RS06]. More recently, the SRT quotient and square root algorithms have been verified with a focus on obtaining small and correct digit-selection tables in ACL2 [Rus13]. This last work also shows that there are several mistakes in a previous work by Kornerup [Kor05].

During his PhD, Harrison verified a full implementation of the exponential function [Har97, Har00a, Har00b]. He therefore developed a rather high-level formalization of FP arithmetic in HOL Light with the full range of FP values [Har99]. Then Intel hired him and has kept on with formal methods. The Pentium Pro operations have been verified [OZGS99]. The Pentium 4 divider has been verified in the Forte verification environment [KK03]. A full proof of the exponential function, from the circuit to the mathematical function was achieved using HOL4 [AAHTH10].

[3] `http://www.lri.fr/~sboldo/research.html`

Now for a few academic works. A development by Jacobi in PVS is composed of both a low-level specification of the standard and proofs about the correctness of floating-point units [Jac01, Jac02]. This has been implemented in the VAMP processor [JB05]. An appealing method mixes model checking and theorem proving to prove a multiplier [AS95]. Another work proves SRT division circuits with Analytica, a tool written in Mathematica [CGZ99]. A nice survey of hardware formal verification techniques, and not just about FP arithmetic, has been done by Kern and Greenstreet [KG99].

1.4.2 Formal Verification of Floating-Point Algorithms

This is a short section as rather few works mix FP arithmetic and formal proofs. As mentioned below, Harrison has verified algorithms using HOL Light. With others, he has also developed algorithms for computing transcendental functions that take into account the hardware, namely IA-64 [HKST99]. Our work in Coq is detailed in Sections 2.3.1 and 2.3.2.

Some additional references are related to FP arithmetic without being directly FP algorithms. To compute an upper bound for approximation errors, that is to say the difference between a function and its approximating polynomial, the Sollya tool generates HOL Light proofs based on sum-of-squares [CHJL11]. The same problem has been studied in Coq using Taylor models [BJMD+12]. Taylor models were also studied for the proof of the Kepler Conjecture [Zum06].

1.4.3 Program Verification

To get a viewpoint about programs, I refer the reader to the very nice `http://www.informationisbeautiful.net/visualizations/million-lines-of-code/` to see how many lines of code usual programs contain. For example, a pacemaker is less than a hundred thousand lines; Firefox is about 10 million lines; Microsoft Office 2013 is above 40 million lines and Facebook is above 60 million lines. An average modern high-end car software is about 100 million lines of code.

It is therefore not surprising to see so many bugs, even in critical applications: offshore platform, space rendezvous, hardware division bug, Patriot missile, USS Yorktown software, Los Angeles air control system, radiotherapy machine and many others in MacKenzie's book [Mac04]. Research has been done and many methods have been developed to prevent such bugs. The methods considered here mainly belong to the family of static analysis, meaning that the program is proved without running it.

Abstract Interpretation

The main intuition behind abstract interpretation is that we do not need to know the exact state of a program to prove some property: a conservative approximation is sufficient. The theory was developed by Cousot and Cousot in the 70s [CC77]. I will only focus on static analysis tools with a focus on floating-point arithmetic.

Astrée is a C program analyzer using abstract interpretation [CCF+05]. It automatically proves the absence of runtime errors. It was successfully applied to large embedded avionics software generated from synchronous specifications. It uses some domain-aware abstract domains, and more generally octagons for values and some linearization of round-off errors.

Fluctuat is a tool based on abstract interpretation dedicated to the analysis of FP programs [GP06]. It is based on an affine arithmetic abstract domain and was successfully applied

to avionics applications [DGP+09]. It was recently improved towards modularity [GPV12] and to handle unstable tests [GP13].

Another tool called RAICP and based on Fluctuat manages the mix of Fluctuat and constraint programming. Constraint programming is sometimes able to refine approximations when abstract interpretation is not [PMR14].

A related work, but less convincing, concerns the estimation of errors in programs written in Scala [DK11]. It is based on affine arithmetic too and includes non-linear and transcendental functions. But all input values must be given to estimate the error of an execution, which is rather limited.

Deductive Verification

Taking the definition from Filliâtre [Fil11], deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it.

The first question is that of defining the correctness of a program. In other words, what is the program supposed to do and how can we express it formally? This is done using a specification language. This is made of two parts: a mathematical language to express the specification and its integration with a programming language. The use of a specification language is needed by deductive verification, but can also be used for documenting the program behavior, for guiding the implementation, and for facilitating the agreement between teams of programmers in modular development of software [HLL+12]. The specification language expresses preconditions, postconditions, invariants, assertions, and so on. This relies on Hoare's work [Hoa69] introducing the concept known today as Hoare triple. In modern notation, if P is a precondition, Q a postcondition and s a program statement, then the triple $\{P\}s\{Q\}$ means that the execution of s in any state satisfying P must end in a state satisfying Q. If s is also required to terminate, this is total correctness, the opposite being partial correctness. Modern tools use weakest preconditions [Dij75]: given s and Q, they compute the weakest requirement over the initial state such that the execution of s will end up in a final state satisfying Q. There is left to check that P implies this weakest requirement. These tools are called verification condition generators and efficient tools exist.

The ACSL specification language for C and the Why3 verification condition generator will be described in the next chapter. As for the others, we will sort them by input language.

Another C verification environment is VCC [CDH+09]. It is designed for low-level concurrent system code written in C. Programs must be annotated in a language similar to ACSL and are translated into an intermediate language called Boogie. VCC includes tools for monitoring proof attempts and constructing partial counterexample executions for failed proofs. It was designed to verify the Microsoft Hyper-V hypervisor. A hypervisor is the software that sits between the hardware and one or more operating systems; the Microsoft Hyper-V hypervisor consists of 60,000 lines of C and assembly code.

Java programs can be annotated using JML, a specification language common to numerous tools [BCC+05]. It has two main usages: the first usage is runtime assertion checking and testing: the assertions are executed at runtime and violations are reported. This also allows unit testing. The second usage is static verification. JML supports FP numbers and arithmetic, but no real numbers are available in the specification language. It has also been noted that NaNs were a pain as far as specification are concerned [Lea06]. Several tools exist, with different degrees of rigor and automation. ESC/Java2 automatically gives a list of possible errors, such as null pointer dereferencing. KeY also generates proof obligations to be discharged by its own

prover integrating automated and interactive proving [ABH+07, BHS07]. A limit is that the user cannot add his own external decision procedures. Jahob is the tool nearest to those used here: it allows higher-order specifications and calls to a variety of provers, both automatic and interactive (Isabelle and Coq). Nevertheless, its philosophy is that interactive provers should not be used: Jahob provides various constructions, such as instantiating lemmas or producing witnesses, to help automated provers. The idea is to have a single environment for programming and proving, but this means heavy annotations (even longer than ours) in the integrated proof language [ZKR09]. Some tools, such as KeY, have the Java Card restrictions and therefore do not support FP arithmetic.

VeriFast is a separation logic-based program verifier for both Java and C programs. The user writes annotations, that are automatically proved using symbolic execution and SMT provers. Its focus is on proving safety properties, such as the absence of memory safety bugs and of race conditions. It has found many such bugs on industrial case studies [PMP+14].

Spec# is an object-oriented language with specifications, defined as a superset of C# [BLS05]. Spec# makes both runtime checking and static verification available, the latest being based on the Boogie intermediate language. However, Spec# lacks several constructs: mathematical specification, ghost variables. Moreover, quantifiers are restricted to the ones that are executable.

Dafny is both an imperative, class-based language that supports generic classes and dynamic allocation and a verifier for this language [Lei10, HLQ12]. The basic idea is that the full verification of a program's functional correctness is done by automated tools. The specification language also includes user-defined mathematical functions and ghost variables. As in VCC and Spec#, Dafny programs are translated into an intermediate language called Boogie. To be executed, the Dafny code may be compiled into .NET code.

The B method uses the same language in specification, design, and program. It depends on set theory and first-order logic and is based on the notion of refinement [Abr05]. It was designed to verify large industrial applications and was successfully applied to the Paris Métro Line 14.

A last work has been done on a toy language with a focus on FP arithmetic [PA13]. As in JML, no real numbers are available and rounding does occur in annotations. The specification language focuses on the stability of computations, but this only means that the computation behaves as in extended precision. This unfortunately means neither that the computation is always stable, nor that it is accurate.

Complementary Approaches

This section describes other approaches that are not directly related to program verification, but have other advantages.

First, interval arithmetic is not a software to verify a program, but a modification of a program. Instead of one FP number, we keep an upper and a lower bound of the real value. Taking advantage of the directed roundings, it is possible to get a final interval that is guaranteed to contain the correct mathematical value. The final interval may be useless, such as $(-\infty, +\infty)$, but the motto is "Thou shall not lie". Large intervals may be due to instability or to the dependency problem. Nevertheless, interval arithmetic is fast and able to prove mathematical properties [Tuc99].

Test is a common validation method, but is seldom used in FP arithmetic as a complete verification method. The reason is very simple: it rapidly blows up. Consider for example a

function with only three `binary64` inputs. There are 2^{192} possibilities. If each possibility takes 1 millisecond to check, an exhaustive test requires more than 10^{47} years. This has nevertheless been used for the table maker's dilemma for one-input functions, using smart filters and optimizations [LM01]. Another work involves automatic test and symbolic execution: it tries to find FP inputs in order to test all execution paths [BGM06].

Another interesting approach is that of certificates: instead of proving *a priori* that a program is correct, the program generates both a result and an object that proves the correctness of the result. An *a posteriori* checking is needed for all calls to the function, but it is very useful in many cases. A good survey has been done by McConnell *et al.* [MMNS11], but the FP examples are not convincing. It seems FP computations are not well-matched with certificates.

An interesting recent work uses dynamic program analysis to find accuracy problem [BHH12]. As in [PA13], the idea is only to compare with a higher precision, here 120 bits. Besides from being slow, this does not really prove the accuracy of the result; it only detects some instabilities.

An orthogonal approach is to modify the program into an algebraically equivalent program, only more accurate. This has been done using Fluctuat and a smart data structure called Abstract Program Expression Graph (APEG) so that it is not required to exhaustively generate all possible transformations in order to compare them [IM13].

Most previous approaches overestimate round-off errors in order to get a correct bound. Using a heuristic search algorithm, it is possible to find inputs that create a large floating-point error [CGRS14]. It will not always produce the worst case, but such a tool could validate the obtained bounds to warrant they are not too much overestimated.

1.4.4 Numerical Analysis Programs Verification

One of my application domains is numerical analysis. Here are some previous works regarding the verification of such algorithms/programs.

Berz and Makino have developed rigorous self-verified integrators for flows of ODEs (Ordinary Differential Equations) based on Taylor models and differential algebraic methods, that are the base of the COSY tool. Some current applications of their methods include proving stability of large particle accelerators, dynamics of flows in the solar system, and computer assisted proofs in hyperbolic dynamics [BM98, BMK06].

A comparable work about the integration of ODEs uses interval arithmetic and Taylor models [NJN07]; it is also a good reference for verified integration using interval arithmetic.

Euler's method is embedded in the Isabelle/HOL proof assistant using arbitrary-precision floating-point numbers [IH12, Imm14]. The Picard operator is used for solving ODEs in Coq using constructive reals [MS13].

Outline After this introduction, the methods and tools used afterwards are described in Chapter 2. The biggest chapter is then Chapter 3: it presents a gallery of formally verified C programs. All programs are motivated, then both data about the proofs and the full annotated C code are given. Chapter 4 emphasizes compilation issues and several solutions. Conclusions and perspectives are given in Chapter 5.

2. Methods and Tools for Floating-Point Program Verification

Here is the programming by contract method for verifying a C program, that can also be found in Figure 2:

- annotate the program: in C special comments, write what the program expects (such as a positive value, an integer smaller than 42) and what the program ensures (such as a result within a small distance of the expected mathematical value). You may also write assertions, loop variants (to prove termination), and invariants (to handle what is happening inside loops).
- generate the corresponding verification conditions (VCs): here I use Frama-C, its Jessie plugin, and Why. This chain acts as a program that takes the previous C program as input and generates a bunch of theorems to prove. If all the theorems are proved, then the program is guaranteed to fulfill its specification and not to crash (due to exceptional behaviors, out-of-bound accesses, and so on).
- prove the VCs. This can be done either automatically or interactively using a proof assistant.

In practice, this is an iterative process: when the VCs are impossible to prove, we modify the annotations, for example strengthen the loop invariant, add assertions or preconditions, until everything is proved.

Note that the first step is critical: if you put too many preconditions (ultimately false), then the function cannot be reasonably used; if you put too few postconditions (ultimately true), then what you prove is useless. This is not entirely true, as a proved empty postcondition means at least that there is no runtime error, which is an interesting result *per se*.

Let us go into more details. This chapter is divided into three sections. Section 2.1 describes the annotation language and the resulting VCs. Section 2.2 introduces the provers. Section 2.3 presents the formalizations of floating-point arithmetic for interactive provers. Section 2.4 describes a Coq library for real analysis.

2.1 Annotation Language and Verification Conditions for Floating-Point Arithmetic

An important work is the design of the annotation language for floating-point arithmetic [BF07]. The goal is to write annotations that are both easy to understand and useful. First, I need to convince the reader of what is proved. Therefore the language must be understandable to those who are not experts in formal methods: I want computer arithmeticians to understand what is proved from the annotations. Second, I need people to use the annotations. Therefore they must correspond to what they want to guarantee on programs.

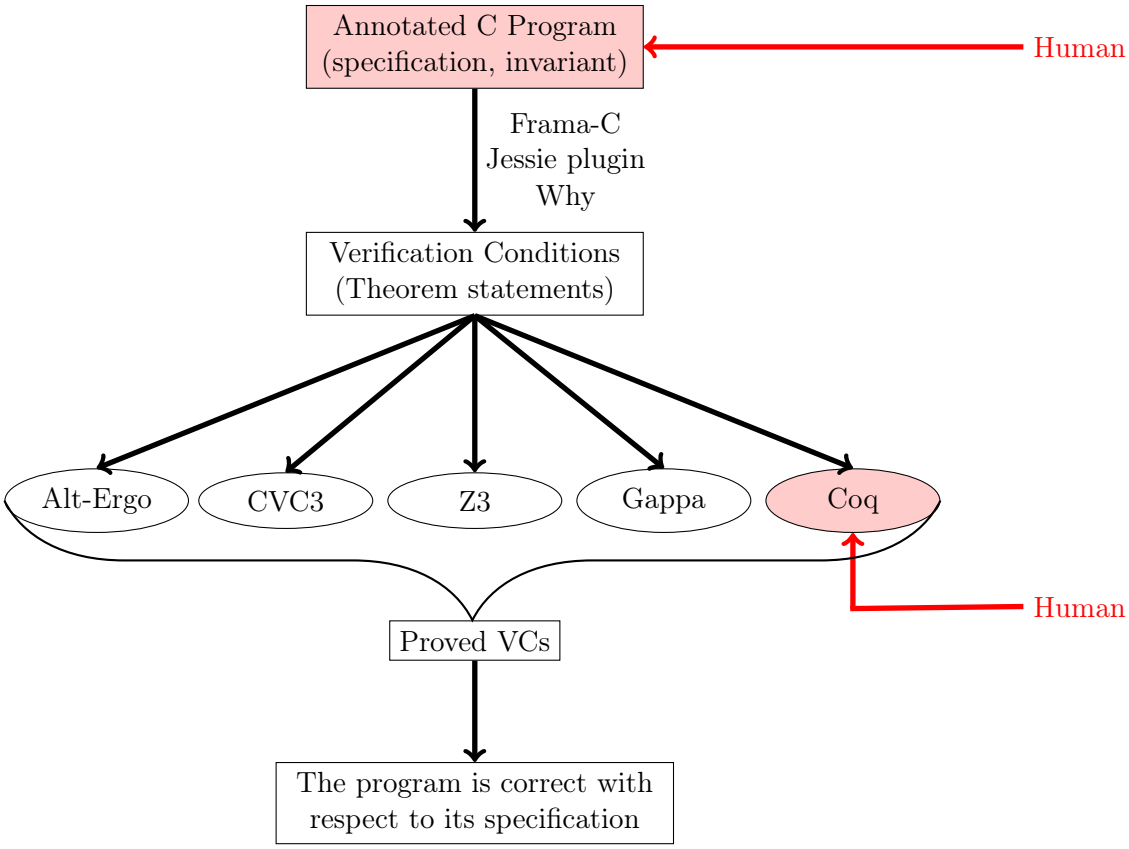

Figure 2.1: Scheme for verifying a C program. Red means the need for human interaction.

With Filliâtre, I first designed this language for Caduceus [FM04, FM07]. Caduceus is a tool for the static analysis of C programs using a weakest precondition calculus developed by Filliâtre and Marché. Then Caduceus was superseded by Frama-C [CKK+12]: it is a framework that takes annotated C as input. The annotation language is called ACSL (for ANSI/ISO C Specification Language) [BCF+14a] and is quite similar to JML: it supports quantifiers and ghost code [FGP14] (model fields and methods in JML) that are for specification only, and therefore not executed. Numerous Frama-C plugins are available for different kinds of analysis such as value analysis using abstract interpretation, browsing of the dataflow, code transformation, and test-case generation. I use deductive verification based on weakest precondition computation techniques. The available plugins are WP and Jessie [Mar07].

The annotation language for floating-point arithmetic was directly taken from Caduceus to Frama-C. The choice is to consider a floating-point number f as a triple of three values:

- its floating-point value: what you really have and compute with in your computer. For intuitive annotations, this value is produced when f is written inside the annotations.
- its exact value, denoted by `\exact(f)`. It is the value that would have been obtained if real arithmetic was used instead of floating-point arithmetic. It means no rounding and no exceptional behavior.
- its model value, denoted by `\model(f)`. It is the value that the programmer intends to compute. This is different from the previous one as the programmer may want to compute an infinite sum or have discretization errors.

A meaningful example is that of the computation of $\exp(x)$ for a small x. The floating-point value is `1+x+x*x/2` computed in `binary64`, the exact value is $1+x+\frac{x^2}{2}$ and the model value is $\exp(x)$. This allows a simple expression of the rounding error: $|f - \texttt{\textbackslash exact}(f)|$ and of the total error. An important point is that, in the annotations, arithmetic operations are mathematical, not machine operations. In particular, there is no rounding error. Simply speaking, we can say that floating-point computations are reflected within specifications as real computations.

Time has shown that the exact value is very useful. It allows one not to state again part of the program in the annotations, which is quite satisfactory. This idea of comparing FP with real computations is caught again by Darulova and Kuncak [DK14]. As for the model value, this is not so clear: people need to express the model part, but they do not use the model value for it. They define the intended value as a logical function and compare it to the floating-point result. In my opinion, the reason is mostly sociological: the annotations then clearly state what the ideal value is. When the model part is set somewhere in the program, it is less convincing. A solution is to express the value of the model part as a postcondition, but it makes the annotations more cumbersome.

The C annotated program is then given to Frama-C and its Jessie plugin. They transform it into an intermediate language called Why. FP numbers and arithmetic are not built-in in Why. The floating-point operations need to be specified in this intermediate language to give meaning to the floating-point operations. This description is twofold. First, it formalizes the floating-point arithmetic with a few results. For example, the fact that all `binary64` roundings are monotonic is stated as follows.

```
axiom round_double_monotonic:
  forall x,y:real. forall m:mode.
   x <= y -> round_double(m,x) <= round_double(m,y)
```

This is an axiom in Why, but all such axioms are proved in Coq using the formalizations described in Section 2.3. This is used by automatic provers, even if they know nothing about FP arithmetic.

The second part of the description is the formalization of floating-point operations. For instance, the addition postcondition states that, if there is no overflow, the floating-point value is the rounding of $x + y$, the exact part is the exact addition of the exact parts of x and y, and similarly for the model part.

There is a choice here: by default, we want to prove that no exceptional behaviors occurs. This means that every addition creates a VC that requires that it does not overflow, and similarly for subtraction, multiplication, and division (with another for the fact that the divisor is non-zero). The idea is to absolutely prevent exceptional behavior. An extension has been done that allows exceptional values and computations on them [AM10]. It can be useful in some cases as computations with infinities is quite reasonable, but it means more possibilities for each computation. Therefore, interactive proofs become incredibly cumbersome. Moreover, our choice implies that inputs of functions cannot be infinities or NaNs. This removes the problems of NaN inputs in annotations described by Leavens [Lea06].

Using weakest preconditions computations, VCs are generated corresponding to FP requirements and giving FP properties. Why is then able to translate a VC, meaning a logical statement into the input language of many provers described in the next section.

There is a subtlety here as two different Why tools are used. First Why2 (that is to say Why 2.x) [Fil03, FM04] is used in most programs. It has a limited ability to call interactive provers (such as Coq or PVS). In this case, all VCs are put into a single file. It means that if VCs 1, 3, and 4 are proved with an automatic prover, but 2 and 5 require interactive proofs, Why2 generates a single file with all 5 VCs. If the user does not want to interactively prove 1, 3, and 4, he may remove it from his file so that only 2 and 5 are left. This is the case only in the example of Section 3.9. But there is no check by an automatic tool that this removing is safe. The next version is Why3 [Fil13] that allows to use a prover on a VC and to check independently the correctness of each VC [BFM+13]. There is therefore no manual removing and this increases the guarantee. Why2 and Why comes with a graphical interface to call any prover on any VC. A screenshot is given in Figure 2.2.

Why3 files include declaration of types and symbols, definitions, axioms, and goals. Such declarations are organized in small components called theories. A theory may have an axiom that states that $0 = 1$. To ensure the consistency of a theory, this theory can be realized: it means that definitions are given and axioms proved, using for instance an interactive prover. This is possible when using only Why3, but not when using the Frama-C/Jessie/Why3 chain for now. This means that axiomatics defined in ACSL (such as the definition of the ulp) need Why2 to be realized. The need for these axiomatics can be questioned, but this seemed the best choice for readability when specifying these programs.

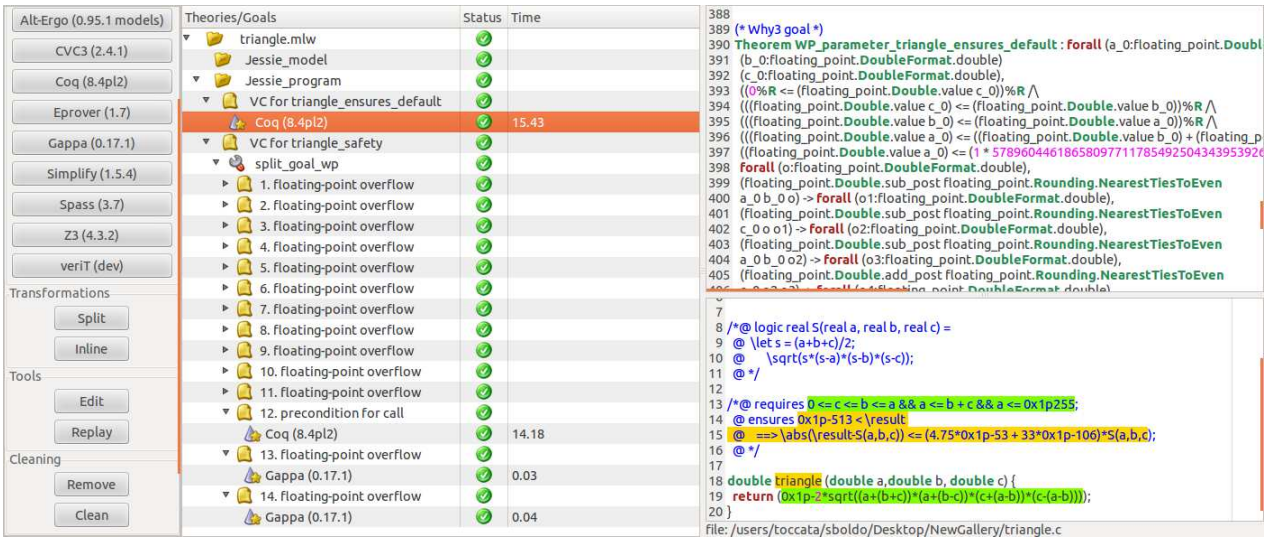

Figure 2.2: Snapshot of the Why3 GUI on the program of Section 3.5.

2.2 Provers

Once the VCs are computed, they must be proved. Several automated provers exist, depending on the underlying logic and theories involved. The Why tools translate the VCs into the input format of the chosen solver.

The solvers that have improved the most recently are SMT solvers. SMT solvers, for Satisfiability Modulo Theories, are tools that decide if a theorem is correct or not. The logic these solvers are based on is usually first-order classical logic, with several theories added, such as arithmetic, tableaux, and bits vectors. The ones used here are Alt-Ergo [CCKL08, Con12], CVC3 [BT07], and Z3 [dMB08]. None of them know anything about FP arithmetic by default. This is the reason for providing them with the preceding description in Why that serves as theory. An efficient solution is bit blasting, even if it may get intractable [AEF$^+$05]. Another SMT solver is MathSat5 [HGBK12], and it has a decision procedure for FP which is smarter than bit blasting. It mainly handles ranges using conflict-driven clause learning. Note there is a standardization of the FP arithmetic for all SMT solvers, that includes exceptional values [RW10].

The prover I use the most is Gappa [DM10, dDLM11]. It is based on interval arithmetic and aims at proving properties of numerical programs, especially ranges of values and bounds on rounding errors. The inputs are logical formulas quantified over real numbers. The atoms are explicit numerical intervals for expressions. Of course, rounding functions may be used inside expressions. The user may provide hints, such as splitting on a variable, to help Gappa succeed. I use Gappa for simple proofs for overflow, for rounding errors, and for a local error in the last program.

Gappa is not only a prover. It is also able to generate a Coq proof of the proved properties, to increase the trust in this yes/no answer. This feature has served to develop a Coq tactic for automatically solving goals related to floating-point and real arithmetic [BFM09].

There is improvement left in these solvers as some VCs may only be proved by combining the features of both Gappa and SMT provers. Some work has been done to mix Gappa and Alt-Ergo [CMRI12], but it is only a prototype.

2.3 Formalizations of Floating-Point Arithmetic

When automatic provers are unable to prove the VCs, which is often the case in the described programs as they are tricky, we are left with interactive provers. Floating-point arithmetic is specified in Why, but must also be formalized in the proof assistants.

While at NIA, I worked with PVS [ORS92] and I defined the high-level floating point arithmetic formalization of the PVS NASA library [BM06] with a few applications. A low-level formalization of the IEEE-754 standard exist [Min95, CM95] and I also related both.

I mainly work with the Coq proof assistant. It comes with a standard library, so that users do not have to start their proofs from scratch but can instead reuse well-known theorems that have already been formally proved beforehand. This general-purpose library contains various developments and axiomatizations about sets, lists, sorting, arithmetic, real numbers, etc. Here, we mainly use the Reals standard library [May01] and we are also working on its improvement (see Section 2.4). A very useful tactic is Interval that simplifies the proofs of inequalities on expressions of real numbers [Mel12].

The standard library does not come with a formalization of floating-point numbers. For that purpose, I have used the two libraries described below.

2.3.1 PFF

The PFF library was initiated by Daumas, Rideau, and Théry [DRT01] in 2001. After a number of proofs during my PhD, I took over from them. I have contributed and maintained the library since then.

Given a radix β greater than one, a FP number x is only a record composed of two signed integers: the mantissa n_x and the exponent e_x. As expected, its real value is obtained by computing $n_x \times \beta^{e_x}$.

An FP format is a couple (N, E_i) of natural numbers that correspond to a bound on the mantissa and to the minimal exponent. An FP number (n, e) is in the format if and only if

$$|n| < N \quad \text{and} \quad -E_i \leq e.$$

In practice, N will be β^p most of the time, with p being the precision of the format. Some extensions have been made to handle exceptional values (infinities and NaNs), but they were unpractical [Bol04a]. Note that several bounded floats will represent the same real value. Consider for example in radix 2 the value: $1 = (1, 0) = (10_2, -1) = (100_2, -2)$. If needed, a canonical value can be computed: it either has the minimal exponent or a mantissa greater or equal to β^{p-1}. In practice, this was seldom used because seldom needed, except for defining the ulp.

The main characteristic of this formalization is the fact that roundings are axiomatically formalized. A rounding here is a property between a real number and a FP number (as expected, it also takes the radix and the format). Contrary to many formalizations, it is not a function from reals to FP numbers. It means in particular that there is no computational content and that several FP numbers may be correct roundings of a real. Moreover, the FP number has to be given, with its proof of being one of the possible rounding. This has advantages: results that hold for any rounding or any rounding to nearest (whatever the tie) are easy to handle.

A drawback is the unintuitive definition of a rounding: it is a property such that any real can be rounded; if a FP number is a rounding of a real value, then any FP number equal to

the first one also has the property of being a rounding of the real number; any rounding is either the rounding up or down of the real number; the rounding is non-decreasing. Of course, IEEE-754 roundings are defined. Another rounding is defined that captures both possible FP numbers when the real to round is in the middle.

Another drawback is the lack of automation: to prove that a floating-point number is correctly rounded is very long and tedious. These choices done, a large library was developed, with many results [Bol04b, BD01, BD02, BD03, BD04a, BD04b, Bol06a, BM08, RZBM09, BDL09, BM11b].

2.3.2 Flocq

Flocq was developed with Melquiond to supersede both PFF, the Coq library of Gappa, and Interval. This section gives the main features of the library [BM11a].

The first feature is that FP numbers are a subset $\mathbb{F}$ of the real numbers. This means there is no requirement for now that $\mathbb{F}$ be discrete or that $0 \in \mathbb{F}$. It therefore removes the need for a coercion from $\mathbb{F}$ to $\mathbb{R}$, which is rather painful in PFF.

There will be two definitions of the rounding modes: one axiomatic as before, but also one computable. A rounding predicate Q has the Coq type $\mathbb{R} \to \mathbb{R} \to \text{Prop}$. For it to be a rounding predicate, it must fulfill two requirements. First, it must be total: any real can be rounded. Second, it must be monotone: $\forall x, y, f, g \in \mathbb{R},\ Q(x, f) \Rightarrow Q(y, g) \Rightarrow x \leq y \Rightarrow f \leq g$ (that is to say nondecreasing). These two properties are sufficient for a rounding predicate to have reasonable properties. For example, those properties imply the uniqueness of rounding: two reals that are roundings of the same value are equal. This is different from PFF where the corresponding property was $x < y \Rightarrow f \leq g$. Indeed, unicity is not required by PFF.

As in PFF (see above), we can then define the common rounding modes by their mathematical properties for a given $\mathbb{F}$. For instance, rounding toward $-\infty$ is defined by $f \in \mathbb{F} \wedge f \leq x \wedge (\forall g \in \mathbb{F}, g \leq x \Rightarrow g \leq f)$.

$\mathbb{F}$ cannot just be any subset of $\mathbb{R}$. Consider for example the set of rational numbers, then an irrational number x cannot be rounded toward $-\infty$: there is always another rational that is smaller than x and nearer to x. To define rounding to nearest, with ties to even, an additional hypothesis on $\mathbb{F}$ is required, so that, among two successive FP numbers, one and only one is even.

Representing formats as predicates and rounding modes as relations on real numbers makes it simple to manipulate them when rounded values are known beforehand. But they are a pain when a proof actually needs to account for a value being the correct rounding. Therefore we have chosen a more computational approach for another representation of formats and rounding modes.

All the formats of this family (called *generic format*) satisfy the following two main properties: they contain only floating-point numbers $m \cdot \beta^e$ and all the representable numbers in a given slice are equally distributed.

The slice of a real number x is given by its discrete β-logarithm $e = \text{slice}(x)$ such that $\beta^{e-1} \leq |x| < \beta^e$. Then, a format $\mathbb{F}_\varphi$ is entirely described by a function $\varphi : \mathbb{Z} \to \mathbb{Z}$ that transforms numbers' discrete logarithms into canonical exponents for this format. In other words, a number x is in $\mathbb{F}_\varphi$ if and only if it can be represented as a floating-point number $m \cdot \beta^{\varphi(\text{slice}(x))}$.

More precisely, the format $\mathbb{F}_\varphi$ is defined as the set of real numbers x such that x is equal to $\mathcal{Z}\left(x \cdot \beta^{-\varphi(\text{slice}(x))}\right) \cdot \beta^{\varphi(\text{slice}(x))}$ with $\mathcal{Z}$ the integer part (rounded toward zero).

As expected, φ must satisfy a few constraints in order to be a valid format with definable roundings. The common formats are fixed-point (FIX), floating-point with unbounded exponents (FLX), floating-point with gradual underflow (FLT, the only available in PFF), and floating-point with abrupt underflow (FTZ). The table below formally defines these formats and how they can be defined using φ functions:

Format	is defined by $\exists f,\ \mathrm{F2R}(f) = x\ \wedge \ldots$	$\varphi(e) =$
$\mathrm{FIX}_{e_{\min}}(x)$	$e_f = e_{\min}$	$e_{\min}$
$\mathrm{FLX}_{p}(x)$	$\lvert n_f\rvert < \beta^p$	$e - p$
$\mathrm{FLT}_{p,e_{\min}}(x)$	$e_{\min} \le e_f \wedge \lvert n_f\rvert < \beta^p$	$\max(e-p, e_{\min})$
$\mathrm{FTZ}_{p,e_{\min}}(x)$	$x \neq 0 \Rightarrow e_{\min} \le e_f \wedge \beta^{p-1} \le \lvert n_f\rvert < \beta^p$	$\begin{cases} e-p & \text{if } e-p \ge e_{\min}, \\ e_{\min}+p-1 & \text{otherwise.} \end{cases}$

Figure 2.3 shows the graph of the corresponding φ functions for the three usual families of floating-point formats.

The *ulp* (unit in the last place) of a real number x is defined as $\beta^{\varphi(\mathrm{slice}(x))}$. This function is partial: it is not defined for 0 (neither was $\mathrm{slice}(x)$). But it is defined for any other real, be it in $\mathbb{F}_\varphi$ or not. Several developments have been done using Flocq, including some I did not contribute to [MDMM13].

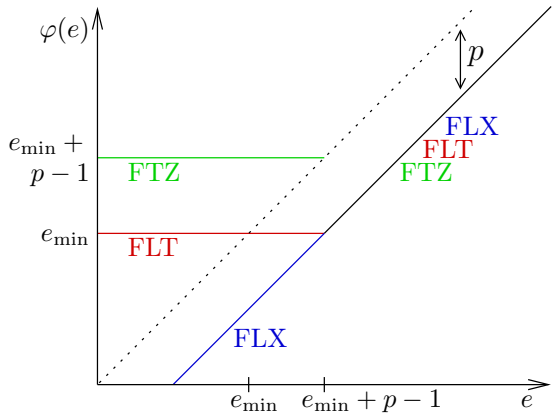

Figure 2.3: Values of φ for formats FLX, FLT, and FTZ, with precision p. These functions are the same for normal numbers ($e \ge e_{\min} + p$), but they diverge for subnormal numbers. In particular, the φ function for FTZ is discontinuous.

FP libraries are a base of this work, but recent applications about numerical analysis have required a formal library about real analysis. Coq has evolved a lot during the last decade, but the standard library for real numbers was designed more than ten years ago and has not evolved with Coq.

2.4 A Real Analysis Coq Library

The limits of the standard library for real numbers was especially glaring when we considered numerical analysis programs and especially the program solving the one-dimensional wave equation of Section 3.9. We wanted to prove that the following d'Alembert's formula

$$\frac{1}{2}\left(u_0(x+ct)+u_0(x-ct)\right)+\frac{1}{2c}\int_{x-ct}^{x+ct} u_1(\xi)\,d\xi+\frac{1}{2c}\int_0^t\int_{x-c(t-\tau)}^{x+c(t-\tau)} f(\xi,\tau)\,d\xi\,d\tau$$

is solution to the partial differential equation of Section 3.9 and regular enough. This was next to impossible using the standard library for real numbers. A reason is the overall use of dependent types: the differentiation operator takes a function f, a real number x, and a proof that f is differentiable at point x, and it returns the value of the derivative of f at x. Similarly, the integration operator requires a proof that the function is integrable. Therefore, taking the second derivative of the previous formula would have been ponderous.

With Lelay and Melquiond, we have worked to develop a user-friendly library of real analysis. After a survey of the main proof assistants and real analysis libraries [BLM14b], we designed the Coquelicot library to increase the practicability for the end user [BLM14a]. We mostly drew our inspiration from Harrison's theory of Euclidean Space [Har13] and from the use of filters and type classes in Isabelle/HOL [HIH13].

We provide definitions for limits, derivatives, integrals, series, and power series [LM12, BLM12]. Whenever different, these definitions are proved equivalent to those of the standard library, so that user developments can easily mix both libraries. For all of those notions of real analysis, total functions are available. These functions return the expected value in case of convergence and an arbitrary infinite or real number otherwise. To help with the proof process, the library comes with a comprehensive set of theorems that cover not only these notions, but also some extensions such as parametric integrals, two-dimensional differentiability, and asymptotic behaviors. A tactic to automate proofs of differentiability is also provided.

3. A Gallery of Verified Floating-Point Programs

Even a seemingly simple numerical algorithm is virtually impossible to analyze with regards to its accuracy. To do so would involve taking into account every single floating-point operation performed throughout the entire computation.

Tucker [Tuc11]

This chapter presents several examples of this "virtually impossible" task: the analysis of the accuracy of FP programs. This is a gallery of FP programs I formally verified using the tools presented in Chapter 2. More details about the proofs can be found in the corresponding papers especially concerning the genericity, such as if the proof holds whatever the radix and what is the required minimal precision. The Coq proofs for the VCs often rely on these generic previous proofs. The size of these proofs are put into a "previous Coq proofs" line to give the reader a better order of magnitude of the full proof. The programs only use `binary64` FP numbers and rounding to nearest, ties to even. For each program, the automated provers, the amount of Coq proofs and the fully annotated C program are given after a short introduction.

The certification of these programs has lasted for years. Therefore tools are not always the same. There is nothing left using Caduceus, but some programs rely on Why3, while others on Why2. This is mainly for historical reasons: the programs proved long ago rely on Why2, while the most recent ones rely on Why3. Another reason is that some results are only available in PFF and that the translation from PFF to Flocq is time-consuming.

Here is an overview of the verified programs. The first ones are basic literature floating-point algorithms: the exact subtraction, also known as Sterbenz's theorem, and Malcolm's algorithm to determine the radix. Then comes Dekker's algorithm to compute the exact error of a multiplication. Then come two algorithms by Kahan, an accurate discriminant and the area of a triangle. The next program KB3D and its variant come from avionics. The last two programs come from numerical analysis: the first one is a simple linear recurrence of order 2. The last one is the biggest piece: it is a numerical scheme that solves the 1D wave equation.

As for the choice of these examples, most of them come from Goldberg's "What every computer scientist should know about floating point arithmetic" [Gol91]. This has been my favorite paper for years as it is both broad, meaning it can be read by non-specialists, and deep, meaning that specialists may learn from it. For me, this is the reference paper that describes what is the "literature" of FP arithmetic, hence my will to formally verify it.

The tool versions used are the ones available at the time this manuscript is written. The number of lines of Coq proofs are given for information only as it may change when upgrading one of the tools. These programs, the list of used tools and their version numbers are available from my web page[1]. All the Coq proofs are also given, so that all the verifications can be run again. By default, the timings are given in seconds on a 3-GHz dual core machine.

[1] `http://www.lri.fr/~sboldo/research.html`

3.1 Exact Subtraction under Sterbenz's Conditions

The most basic property of floating-point arithmetic is probably this one: if two FP numbers are near one to another, their subtraction is exact. Its paternity is usually given to Sterbenz [Ste74] even if Kahan also claims it. More precisely, it states that, if x and y are FP numbers such that $\frac{y}{2} \leq x \leq 2y$, then $x - y$ is computed without error, because it fits into a FP number. This may be a benign cancellation as it is a correct computation. This may also be a catastrophic cancellation, as it magnifies the previous errors. Nevertheless, it is not the subtraction that is inaccurate, even if it is often blamed for inaccuracy.

The program is put on the next page. Note that computations inside annotations are exact. Therefore, the value `x-y` in the postcondition is the mathematical subtraction between `x` and `y`. Two VCs are generated using Why2: one for the postcondition and one for safety, corresponding to the fact that there will be no FP overflow.

Proof obligations	**Coq** Nb lines	Time
VC for behavior	7	
VC for safety	20	
Total (25 lines spec VC excluded)	27	1.93

This program is also verified using Why3. The same VCs are generated, but I take advantage of CVC3 to automatically prove the overflow VC.

Proof obligations	**CVC3**	**Coq**	
			Nb lines
VC for behavior		2.34	6
VC for safety	0.23		

```
/*@ requires  y/2. <= x <= 2.*y;
  @ ensures   \result == x-y;
  @*/

float Sterbenz(float x, float y) {
  return x-y;
}
```

3.2 Malcolm's Algorithm

This is a program known since the 70s that computes the radix [Mal72]. This may seem a strange idea as radix 2 is now omnipresent, but this algorithm runs on any kind of architecture (including pocket calculators). If extended registers are used (see Chapter 4), this algorithm fails as explained by Gentleman and Marovich [GM74]. Note also that radix 10 is back thanks to the new IEEE-754 standard and a few CPU with radix-10 units have already appeared. It may give this algorithm a new boost.

The idea is to get one of the smallest positive values such that $x = \circ(x + 1)$. Then, the smallest FP value greater than x is $x + \beta$. The proof of this algorithm for any radix is rather complicated as the values 2^i may turn inexact due to rounding on radices other than powers of 2. Our formalization for C programs only considers radix-2, therefore we exactly know which and how much computations will be done. In particular, each multiplication by 2 is exact.

The result of this program is proved to be 2, but the most interesting part is the termination of this program and its loop invariant. It may be doubtful that a while loop with condition A != A+1 does terminate.

Proof obligations		**Coq** Nb lines	Time
VC for behavior	1. loop invariant initially holds	3	
	2. loop invariant initially holds	2	
	3. loop invariant initially holds	2	
	4. loop invariant preserved	16	
	5. loop invariant preserved	2	
	6. loop invariant preserved	73	
	7. assertion	48	
	8. assertion	5	
	9. loop invariant initially holds	2	
	10. loop invariant initially holds	2	
	11. loop invariant preserved	9	
	12. loop invariant preserved	13	
	13. postcondition	21	
VC for safety	1. floating-point overflow	9	
	2. floating-point overflow	8	
	3. arithmetic overflow	1	
	4. arithmetic overflow	1	
	5. variant decreases	1	
	6. variant decreases	3	
	7. floating-point overflow	5	
	8. floating-point overflow	7	
	9. floating-point overflow	5	
	10. arithmetic overflow	1	
	11. arithmetic overflow	1	
	12. variant decreases	1	
	13. variant decreases	1	
Total (1,191 lines spec VC excluded)		242	1 min 15

```
/*@ ensures \result == 2.; */

double malcolm1() {
  double A, B;
  A=2.0;
  /*@ ghost int i = 1; */

  /*@ loop invariant A== \pow(2.,i) &&
    @          1 <= i <= 53;
    @ loop variant (53-i); */

  while (A != (A+1)) {
    A*=2.0;
    /*@ ghost i++; */
  }

  /*@ assert i==53 && A== 0x1.p53; */

  B=1;
  /*@ ghost i = 1;*/

 /*@ loop invariant B == i && (i==1 || i == 2);
   @ loop variant (2-i); */

  while ((A+B)-A != B) {
    B++;
    /*@ ghost i++; */
  }

  return B;

}
```

3.3 Dekker's Algorithm

Other algorithms from the 70s are those of Veltkamp and Dekker [Vel69, Dek71]. Veltkamp's algorithm splits a FP number into its most significant half and its least significant part. More precisely, from a p-bit FP number x, it gets its rounding on about $\frac{p}{2}$ bits that is $h_x = \circ_{\frac{p}{2}}(x)$ and the tail t_x such that $x = h_x + t_x$ and t_x also fits in about $\frac{p}{2}$ bits. The "about" is due to the parity of p. When p is even, both h_x and t_x are on $\frac{p}{2}$ bits. In radix 2, they both fit on $\lfloor \frac{p}{2} \rfloor$ bits as the sign of t_x saves us a bit. In the other cases, h_x and t_x may fit on $\lfloor \frac{p}{2} \rfloor$ and $\lceil \frac{p}{2} \rceil$ bits (or vice versa) depending on the chosen constant C.

Then Dekker's algorithm builds upon the previous one to compute the error of the FP multiplication. The idea is to compute the upper and the lower parts of x and y and multiply all the parts without rounding error. Then, the algorithm subtracts from the rounded multiplication all the multiplied parts in the right order, *i.e.* from the largest to the smallest. The result is the correct multiplication subtracted from the rounded one, that is the error.

In order to guarantee that the multiplications of the halves are correct, we have to require the radix to be 2 or the precision to be even [Bol06b]. Underflow is a bit tricky to handle, as the returned value may not be the exact error anymore, but only a reasonable approximation [Bol06b]. The algorithm proof provides the underflow constant 2^{-969}. The overflow constants are such that no overflow will happen, especially in the multiplication by C. These VCs are proved by the gappa tactic, hence the one-line proofs. The first requirement states that the input `xy` is equal to the correct rounding to nearest of `x*y`. This is for the function to have a single FP number as a result, instead of a structure.

Proof obligations		**Coq** Nb lines	Time
Previous Coq proof (spec + proof)		2,639	
VC for behavior	1. assertion	3	
	2. postcondition	238	
VC for safety	1. floating-point overflow	2	
	2. floating-point overflow	1	
	3. floating-point overflow	2	
	4. floating-point overflow	1	
	5. floating-point overflow	1	
	6. floating-point overflow	1	
	7. floating-point overflow	1	
	8. floating-point overflow	1	
	9. floating-point overflow	1	
	10. floating-point overflow	37	
	11. floating-point overflow	47	
	12. floating-point overflow	43	
	13. floating-point overflow	64	
	14. floating-point overflow	43	
	15. floating-point overflow	83	
	16. floating-point overflow	49	
	17. floating-point overflow	94	
Total (1,248 lines spec VC excluded)		3,351	9 min 02

```
/*@ requires xy == \round_double(\NearestEven,x*y) &&
  @          \abs(x) <= 0x1.p995 &&
  @          \abs(y) <= 0x1.p995 &&
  @          \abs(x*y) <=  0x1.p1021;
  @ ensures  ((x*y == 0 || 0x1.p-969 <= \abs(x*y))
  @                ==> x*y == xy+\result);
  @*/
double Dekker(double x, double y, double xy) {

  double C,px,qx,hx,py,qy,hy,tx,ty,r2;
  C=0x8000001p0;
  /*@ assert C == 0x1p27+1; */

  px=x*C;
  qx=x-px;
  hx=px+qx;
  tx=x-hx;

  py=y*C;
  qy=y-py;
  hy=py+qy;
  ty=y-hy;

  r2=-xy+hx*hy;
  r2+=hx*ty;
  r2+=hy*tx;
  r2+=tx*ty;
  return r2;
}
```

3.4 Accurate Discriminant

Computing a discriminant, namely $b \times b - a \times c$ is known to be a FP mess. When the two subtracted values are nearly equal, only the FP errors made during the multiplications are left and the final error may be relatively large. A solution is the following algorithm due to Kahan [Kah04]. It tests whether we are in the cancellation case or not. If not, it goes for the naive algorithm (that is accurate enough). In case of possible cancellation, it computes the errors of both multiplications (using the previous algorithm, with the previously given annotations) and adds all the terms in the correct order to get an accurate result.

The tricky part lies in the test: it is supposed to clearly make apart the cancellation cases from the other cases. In the initial pen-and-paper proof [Kah04, BDKM06], the result of the test was assumed to be correct, meaning as if it were computed without error. Unfortunately, this is not the case due to the multiplication by 3 and this was discovered during the formal verification of the program. Fortunately, this does not change the accuracy of the final result: in the special cases where the FP test may be wrong, we know that $p + q \approx 3 \times |p - q|$ and therefore $b \times b$ and $a \times c$ are about in a factor of 2. This does not suffice to apply Sterbenz's theorem, but it does suffice to bound the cancellation and to get the same error bound, namely 2 ulps.

Note that the ulp is not an ACSL primitive, and therefore needs to be axiomatized. Here is a precise definition of the ulp: it is a power of 2, it is equal to 2^{-1074} for subnormals and it is such that $|f| < 2^{53}\text{ulp}(f)$ otherwise. This set of axioms is required to guarantee that what I call ulp is indeed the FP common value, and not a constant I put to make the proof run, such as 2^{2000}. This axiomatic is then realized in Coq to relate it to the ulp defined in Flocq.

A recent work has improved upon this accuracy using the FMA operator [JLM13].

Proof obligations		**Coq** Nb lines	Time
Previous Coq proof (spec + proof)		3,390	
VC for theory realization		88	
VC for behavior	1. postcondition	61	
	2. postcondition	90	
VC for safety	1. floating-point overflow	2	
	2. floating-point overflow	2	
	3. floating-point overflow	3	
	4. floating-point overflow	4	
	5. floating-point overflow	4	
	6. precondition for call	2	
	7. precondition for call	9	
	8. precondition for call	1	
	9. precondition for user call	2	
	10. precondition for user call	2	
	11. precondition for user call	2	
	12. precondition for user call	2	
	13. precondition for user call	2	
	14. floating-point overflow	44	
	15. floating-point overflow	45	
Total (1,146 lines spec VC excluded)		3,655	5 min 47

```
/*@ axiomatic FP_ulp {
  @ logic real ulp(double f);
  @
  @ axiom ulp_normal1 :
  @   \forall double f; 0x1p-1022 <= \abs(f)
  @       ==>\abs(f) < 0x1.p53 * ulp(f);
  @ axiom ulp_normal2 :
  @   \forall double f; 0x1p-1022 <= \abs(f)
  @       ==> ulp(f) <= 0x1.p-52 * \abs(f);
  @ axiom ulp_subnormal :
  @   \forall double f; \abs(f) <  0x1p-1022
  @       ==> ulp(f) == 0x1.p-1074;
  @ axiom ulp_pow :
  @   \forall double f; \exists integer i;
  @         ulp(f) == \pow(2.,i);
  @ } */


/*@ ensures \result==\abs(f); */
double fabs(double f);


/*@ requires
  @     (b==0.   || 0x1.p-916 <= \abs(b*b)) &&
  @     (a*c==0. || 0x1.p-916 <= \abs(a*c)) &&
  @     \abs(b) <= 0x1.p510 && \abs(a) <= 0x1.p995 && \abs(c) <= 0x1.p995 &&
  @     \abs(a*c) <= 0x1.p1021;
  @ ensures \result==0.  || \abs(\result-(b*b-a*c)) <= 2.*ulp(\result);
  @ */
double discriminant(double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=Dekker(b,b,p);
    dq=Dekker(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

3.5 Area of a Triangle

Another algorithm by Kahan computes the area of a triangle. The common formula is two millenia old and attributed to Heron of Alexandria, but it is inaccurate for degenerated triangles. Kahan's formula is mathematically equivalent, but numerically stable. There is no test needed here, only an algebraically equivalent formula.

This was first "published" on the web [Kah86], and then published in Goldberg's article [Gol91] with a bound of $11\,\varepsilon$, where ε stands for $\frac{\beta^{1-p}}{2}$ that is 2^{-53} in `binary64`. My work improved the error bound to $6.625\,\varepsilon$ [Bol13] and I recently improved it to $4.75\,\varepsilon$.

It seems this example has drawn some academic attention recently. A comparable error bound is proved, but only on very specific triangles, which are non-pathological: $a = 9$, $b = c$ and $4.501 \leq b \leq 4.89$ [DK11] and $b = 4$, $c = 8.5$ and $4.500005 \leq a \leq 6.5$ [DK14]. In the last case, it is additionally required that $b + c > a + 10^{-6}$, in order to prove that the square root will not fail. I proved that this assumption is superfluous. Another work proves that both Kahan's and Heron's formula are unstable [PA13], which is correct, but does not give any insight on the final accuracy of the result.

Proof obligations		**Gappa**	**Coq**	
				Nb lines
Previous Coq proof			18.89	2,091
VC for behavior			16.00	82
VC for safety	1. floating-point overflow	0.02		
	2. floating-point overflow	0.03		
	3. floating-point overflow	0.03		
	4. floating-point overflow	0.03		
	5. floating-point overflow	0.03		
	6. floating-point overflow	0.00		
	7. floating-point overflow	0.02		
	8. floating-point overflow	0.01		
	9. floating-point overflow	0.00		
	10. floating-point overflow	0.02		
	11. floating-point overflow	0.02		
	12. precondition for call		13.22	13
	13. floating-point overflow	0.03		
	14. floating-point overflow	0.04		

```
/*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);



/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */

/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @    ==> \abs(\result-S(a,b,c)) <= (4.75*0x1p-53 + 33*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

3.6 KB3D

This program comes from avionics. More precisely, KB3D takes as inputs the position and velocity vectors of two aircraft. The output is a maneuver for preventing conflicts that involves the modification of a single parameter of the original flight path: vertical speed, heading, or ground speed [DMC05, MSCD05]. This algorithm is distributed: each plane runs it simultaneously and independently. As any avionics application, the certification level is quite high. A formal proof in PVS of the whole algorithm has been done. It is rather long even if the program itself is rather short. But this formal proof was done assuming correct computations on $\mathbb{R}$. Unfortunately, FP inaccuracies may alter the decisions made by this algorithm.

Here, I present a small part of KB3D (rewritten in C) that decides to go on the left or on the right by computing the sign of a quantity. This is typically where FP arithmetic may make results go wrong. The algorithm was therefore modified in order to have the following specification: when it gives an answer, it is the correct one. The idea is to have an error bound on the value to be tested. If the value is over this error bound, it is sure to be positive, regardless of the errors. If the value is under the error bound, it is sure to be negative. In the other cases, the result is of unknown sign and 0 is returned [BN10, BN11]. A common decision should then be taken or a more accurate result should be computed. As for the value of the error bound 2^{-45}, it is the smallest automatically proved by Gappa.

This example shows that the exact value does not mix well with modularity. Here, as s_x (and similarly for v_x, s_y, and v_y) is unknown, it is assumed to be exact. In another context where we would know the relative error bound, this bound could be put as precondition, and the postcondition modified accordingly.

Proof obligations		**Alt-Ergo**	**CVC3**	**Gappa**
VC for behavior of eps_line		0.06		
VC for safety of eps_line	1. floating-point overflow			0.01
	2. floating-point overflow			0.00
	3. floating-point overflow			0.00
	4. floating-point overflow			0.01
	5. floating-point overflow			0.00
	6. precondition for call			0.01
	7. floating-point overflow			0.01
	8. floating-point overflow			0.00
	9. floating-point overflow			0.01
	10. floating-point overflow			0.00
	11. floating-point overflow			0.00
	12. precondition for call			0.02
	13. arithmetic overflow		0.43	
VC for behavior of sign		0.04		
VC for safety of sign				0.00

```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  (\result != 0 ==> \result == l_sign(\exact(x))) &&
  @          \abs(\result) <= 1 ;
  @*/
int sign(double x, double e1, double e2) {

  if (x > e2)
    return 1;
  if (x < e1)
    return -1;
  return 0;
}

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/

int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;

  s1=sign(sx*vx+sy*vy, -0x1p-45, 0x1p-45);
  s2=sign(sx*vy-sy*vx, -0x1p-45, 0x1p-45);

  return s1*s2;
}
```

3.7 KB3D – Architecture Independent

This is nearly the same example as before. The only differences are the pragma at the beginning of the file and the error bound. The pragma means that this program is considered to be possibly compiled on various platforms, possibly with FMA or extended registers and that the postcondition must remain true whatever the compilation choices. See Chapter 4 for more details.

This is important here as KB3D is a coordinated algorithm that will run on several planes. Therefore the decisions must be coherent between the planes and be coordinated in any hardware setting. This means that the algorithm answer must be the same, whatever the underlying processor and compiler.

The main difference with the previous program lies in the error bound, that is slightly greater than in the previous case. As explained in Chapter 4, this will be enough to cover compilation and hardware variety [BN10, BN11]. As before, the value of the error bound is the smallest automatically proved by Gappa.

Proof obligations		**Alt-Ergo**	**CVC3**	**Gappa**
VC for behavior of eps_line		1.34		
VC for safety of eps_line	1. floating-point overflow			0.00
	2. floating-point overflow			0.00
	3. floating-point overflow			0.00
	4. floating-point overflow			0.00
	5. floating-point overflow			0.00
	6. floating-point overflow	0.04		
	7. precondition for call			0.02
	8. floating-point overflow			0.01
	9. floating-point overflow			0.00
	10. floating-point overflow			0.07
	11. floating-point overflow	0.04		
	12. floating-point overflow	0.04		
	13. floating-point overflow	0.04		
	14. precondition for call			0.50
	15. arithmetic overflow		0.67	
VC for behavior of sign		0.04		
VC for safety of sign		0.03		

```
#pragma JessieFloatModel(multirounding)

//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  (\result != 0 ==> \result == l_sign(\exact(x))) &&
  @          \abs(\result) <= 1 ;
  @*/
int sign(double x, double e1, double e2) {

  if (x > e2)
    return 1;
  if (x < e1)
    return -1;
  return 0;
}

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/

int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;

  s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);

  return s1*s2;
}
```

3.8 Linear Recurrence of Order Two

The last two programs come from numerical analysis. This one is an oversimplification of the next one and reduces to computing the n-th term of a sequence defined by a linear recurrence of order 2: $u_{n+1} = 2u_n - u_{n-1}$. In the general case, the value of the sequence is unbounded, therefore the rounding error is unbounded. I choose a particular case where the sequence is bounded (at least its exact part). For the sake of simplicity, I assume that $|u_n| \leq 1$ for all n. As I bound the rounding error, I am able to bound the computed value by 2. This requires one to bound the number of iteration, in this precise case by 2^{26}. I then prove that the exact value of u_n is $u_0 + n \times (u_1 - u_0)$ and that the rounding error of u_n is bounded by $\frac{n(n+1)}{2} 2^{-53}$. This is due to the fact that the rounding errors do compensate and partially cancel each other.

The difficult point is of course the loop invariant: it needs to describe the exact value of the sequence and the precise expression of the rounding error. More precisely, it is based on a predicate `mkp` that is defined only in Coq. If we denote by $\delta(f) = f - \text{exact}(f)$, then the `mkp` predicate states that

$$\begin{array}{l} mkp(u_c, u_p, n) = \exists \varepsilon : \mathbb{N} \to \mathbb{R}, \quad \forall i \in \mathbb{N}, i \leq n \Rightarrow |\varepsilon_i| \leq 2^{-53} \\ \quad \wedge\ \delta(u_p) = \sum_{j=0}^{n-1} (n-j)\ \varepsilon_j + (1-n)\ \delta(u_0) + n\ \delta(u_1) \\ \quad \wedge\ \delta(u_c) = \sum_{j=0}^{n} (n+1-j)\ \varepsilon_j + (-n)\ \delta(u_0) + (n+1)\ \delta(u_1) \end{array} .$$

The existence of this ε can be stated in ACSL, but it would be long and impractical. We therefore prefer to define it using the Coq proof assistant. The exhibit of the error compensation greatly improves the bound on the rounding error.

Proof obligations		**Coq** Nb lines	Time
VC for theory realization		9	
Previous Coq proof about the realization (spec + proof)		228	
VC for behavior	1. loop invariant initially holds	2	
	2. loop invariant initially holds	2	
	3. loop invariant initially holds	3	
	4. loop invariant initially holds	3	
	5. loop invariant initially holds	7	
	6. assertion	24	
	7. loop invariant preserved	2	
	8. loop invariant preserved	2	
	9. loop invariant preserved	9	
	10. loop invariant preserved	4	
	11. loop invariant preserved	229	
	12. postcondition	3	
	13. postcondition	9	
VC for safety	1. floating-point overflow	8	
	2. floating-point overflow	12	
	3. arithmetic overflow	1	
	4. arithmetic overflow	10	
	5. variant decreases	1	
	6. variant decreases	3	
Total (1,230 lines spec VC excluded)		571	20.50

```
/*@ axiomatic MKP {
  @  predicate mkp(double uc, double up, integer n);
  @ } */


/*@ requires 2 <= N <= \pow(2,26) &&
  @      \exact(u0)==u0 && \exact(u1)==u1 &&
  @      \forall integer k; 0 <= k <= N ==>  \abs(u0+k*(u1-u0)) <= 1;
  @ ensures  \exact(\result)==u0+N*(u1-u0) &&
  @          \round_error(\result) <= N*(N+1.)/2.*\pow(2,-53);
  @*/


double comput_seq(double u0, double u1, int N) {
  int i;
  double uprev, ucur,tmp;
  uprev=u0;
  ucur=u1;

  /*@ loop invariant 2 <= i && i <= N+1 &&
    @   \exact(ucur) ==u0+(i-1)*(u1-u0) &&
    @   \exact(uprev)==u0+(i-2)*(u1-u0) &&
    @   mkp(ucur,uprev,i-2);
    @ loop variant N-i; */
  for (i=2; i<=N; i++) {
    tmp=2*ucur;
    /*@ assert tmp==2.*ucur; */
    tmp-=uprev;
    uprev=ucur;
    ucur=tmp;
  }
  return ucur;
}
```

3.9 Numerical Scheme for the One-Dimensional Wave Equation

The last and largest program is another numerical analysis program. This is the result of the Fφst project[2] I led. This was done in collaboration with Clément, Filliâtre, Mayero, Melquiond and Weis. From a mathematical point of view, we try to solve the 1D wave equation, that describes the undulation of a rope. Precisely, we look for p sufficiently regular such that

$$\frac{\partial^2 p(x,t)}{\partial t^2} - c^2 \frac{\partial^2 p(x,t)}{\partial x^2} = s(x,t)$$

and with given values for $p(x,0)$ and $\frac{\partial p(x,0)}{\partial t}$.

To approximate p on a discrete grid, we use a very simple scheme, the second-order centered finite difference scheme. The value $p_j^k \approx p(j\Delta x, k\Delta t)$ is defined by

$$\frac{p_j^k - 2p_j^{k-1} + p_j^{k-2}}{\Delta t^2} - c^2 \frac{p_{j+1}^{k-1} - 2p_j^{k-1} + p_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

and similar formulas for p_i^0 and p_i^1.

This example has been detailed in several articles cited below, therefore technical details are omitted here. The verification of this program can be split in 3 parts:

- the mathematical proof [BCF+10]. Based on a pen-and-paper proof, this is a Coq proof that the numerical scheme approximates the exact mathematical solution with a convergence of order 2. This is a well-known proof in mathematics, but it requires a uniform big O for the proof to be valid. A recent work automatically produces the order of accuracy of a numerical scheme from its C code [ZSR14].

- the bound on the rounding error [Bol09]. Without considering compensation, this value grows to $2^k \times 2^{-53}$. To get a tight error bound, I used the same trick as in the previous example. The rounding errors do compensate and this leads to an error bound proportional to $k^2 \times 2^{-53}$ where k is the number of iterations. As before, we need a complex loop invariant that states the existence of an ε function (from $\mathbb{Z}^2$ to $\mathbb{R}$) that gives the rounding error when computing a single p_i^k. Then, the total error can be expressed as:

  $$p_i^k - exact(p_i^k) = \sum_{l=0}^{k} \sum_{j=-l}^{l} \alpha_j^l \ \varepsilon_{i+j}^{k-l}$$

  with well-chosen constants α_j^l. We assume the mathematical solution is bounded by 1. We deduce that the computed values are bounded by 2 and therefore that the ε_i^k are bounded by the constant 78×2^{-52}. Finally, the rounding error of each computed value can be bounded by $\left|p_i^k - exact\left(p_i^k\right)\right| \leq 78 \times 2^{-53} \times (k+1) \times (k+2)$.

- the full program verification [BCF+13, BCF+14b]. Given the two previous verification steps, there are two checks left to do. The first check concerns all the run-time errors: overflows on integers, overflows on FP numbers, memory accesses. This is a large

[2] http://fost.saclay.inria.fr/

amount of VCs, but most are automatically proved. The last check is putting everything together: it means writing the C annotations, and binding the previous proofs to the generated VCs. This is a time-consuming task. Furthermore, this is the only example where I edited by hand the Coq proof to remove automatically-proved VCs. The handling of the memory was an additional unexpected burden. As the matrix p is modified at each step, I have to handle the cumbersome fact that it is modified at one point only, and that the rest of the matrix (and its properties) are left untouched.

An interesting follow-up of this work is the Coquelicot library described in Section 2.4. The mathematical proof requires that the exact solution is sufficiently regular, in particular near its Taylor polynomial. This can be proved on pen-and-paper, as this exact solution is defined by d'Alembert's formula. But this was impossible to prove and even to state using the standard library for the reals. We therefore put this as an axiom in the annotations. We removed this axiom recently using the Coquelicot library [BLM14a].

The following table describes how the more than 150 VCs are proved, generated by the 166 lines of comments and 32 lines of C. Gray cells describe when the automatic prover fails (in a given time); checkmarks in dark green cells describe when the automatic prover is able to prove the VC.

Proof obligations		**Alt-Ergo**	**CVC3**	**Gappa**	**Z3**	**Coq** Lines	**Coq** Time
Previous Coq proof about convergence						4,108	59.71
Previous Coq proof about FP error						1,623	1 min 07
VC for theory	Theory realization					80	
	1. Lemma alpha_conv_pos					17	
	2. Lemma analytic_error_le					244	
	3. Lemma sqr_norm_dx_conv_err_0					5	
	4. Lemma sqr_norm_dx_conv_err_succ					12	
	Other lemmas about realization					1,595	
	Total for VC realization & lemmas					1,953	20 min 18
VC for behavior	1. assertion			✓			
	2. assertion			✓			
	3. assertion					113	
	4. assertion					5	
	5. assertion					35	
	6. assertion					14	
	7. assertion					26	
	8. assertion					40	
	9. loop invariant initially holds	✓	✓		✓		
	10. loop invariant initially holds	✓	✓		✓		
	11. loop invariant initially holds					4	
	12. loop invariant preserved	✓	✓		✓		
	13. loop invariant preserved	✓	✓		✓		

	14. loop invariant preserved					22
	15. assertion					6
	16. loop invariant initially holds	✓	✓		✓	
	17. loop invariant initially holds	✓	✓		✓	
	18. loop invariant initially holds					6
	19. assertion				✓	
	20. assertion				✓	
	21. assertion				✓	
	22. loop invariant preserved	✓	✓		✓	
	23. loop invariant preserved	✓	✓		✓	
	24. loop invariant preserved					273
	25. assertion					6
	26. loop invariant initially holds	✓	✓		✓	
	27. loop invariant initially holds	✓	✓		✓	
	28. loop invariant initially holds	✓	✓		✓	
	29. loop invariant initially holds	✓	✓		✓	
	30. loop invariant initially holds	✓	✓		✓	
	31. loop invariant initially holds					5
	32. assertion					3
	33. assertion					3
	34. assertion					3
	35. assertion					3
	36. loop invariant preserved	✓	✓		✓	
	37. loop invariant preserved	✓	✓		✓	
	38. loop invariant preserved					238
	39. assertion					6
	40. loop invariant preserved	✓	✓		✓	
	41. loop invariant preserved	✓	✓		✓	
	42. loop invariant preserved	✓	✓		✓	
	43. postcondition					19
	44. postcondition					115
VC for safety	1. floating-point overflow			✓		
	2. floating-point overflow	✓	✓	✓		
	3. floating-point overflow			✓		
	4. floating-point overflow	✓	✓	✓	✓	
	5. floating-point overflow					71
	6. floating-point overflow					48
	7. floating-point overflow					46
	8. arithmetic overflow	✓	✓	✓	✓	
	9. arithmetic overflow	✓	✓	✓	✓	
	10. arithmetic overflow	✓	✓	✓	✓	
	11. arithmetic overflow	✓	✓	✓	✓	
	12. precondition for user call	✓	✓	✓	✓	
	13. precondition for user call	✓	✓	✓	✓	
	14. pointer dereferencing	✓	✓		✓	

15. pointer dereferencing	✓	✓		✓	
16. pointer dereferencing		✓			
17. pointer dereferencing		✓			
18. floating-point overflow					12
19. floating-point overflow			✓		
20. precondition for user call	✓	✓	✓	✓	
21. precondition for user call					213
22. pointer dereferencing	✓	✓		✓	
23. pointer dereferencing	✓	✓		✓	
24. pointer dereferencing	✓	✓			
25. pointer dereferencing	✓	✓			
26. arithmetic overflow	✓	✓		✓	
27. arithmetic overflow	✓	✓		✓	
28. variant decreases	✓	✓		✓	
29. variant decreases	✓	✓		✓	
30. pointer dereferencing	✓	✓		✓	
31. pointer dereferencing	✓	✓		✓	
32. pointer dereferencing	✓	✓		✓	
33. pointer dereferencing	✓	✓		✓	
34. pointer dereferencing	✓			✓	
35. pointer dereferencing		✓			
36. arithmetic overflow	✓	✓		✓	
37. arithmetic overflow	✓	✓		✓	
38. pointer dereferencing	✓	✓		✓	
39. pointer dereferencing	✓	✓		✓	
40. pointer dereferencing	✓	✓			
41. pointer dereferencing	✓	✓			
42. pointer dereferencing	✓	✓		✓	
43. pointer dereferencing	✓	✓		✓	
44. pointer dereferencing	✓	✓			
45. pointer dereferencing	✓	✓			
46. floating-point overflow		✓			
47. floating-point overflow					6
48. arithmetic overflow	✓	✓		✓	
49. arithmetic overflow	✓	✓		✓	
50. pointer dereferencing	✓	✓		✓	
51. pointer dereferencing	✓	✓		✓	
52. pointer dereferencing	✓	✓			
53. pointer dereferencing	✓	✓			
54. floating-point overflow					7
55. pointer dereferencing	✓	✓		✓	
56. pointer dereferencing	✓	✓		✓	
57. floating-point overflow		✓	✓		
58. floating-point overflow			✓		
59. floating-point overflow					14

60. pointer dereferencing	✓	✓		✓	
61. pointer dereferencing		✓			
62. variant decreases	✓	✓	✓	✓	
63. variant decreases	✓	✓		✓	
64. pointer dereferencing	✓	✓		✓	
65. pointer dereferencing		✓		✓	
66. arithmetic overflow	✓	✓		✓	
67. arithmetic overflow	✓	✓		✓	
68. pointer dereferencing	✓	✓		✓	
69. pointer dereferencing		✓			
70. arithmetic overflow	✓	✓		✓	
71. arithmetic overflow	✓	✓		✓	
72. pointer dereferencing	✓	✓		✓	
73. pointer dereferencing	✓	✓		✓	
74. pointer dereferencing	✓	✓			
75. pointer dereferencing	✓	✓		✓	
76. pointer dereferencing	✓	✓		✓	
77. pointer dereferencing	✓	✓		✓	
78. pointer dereferencing	✓	✓			
79. pointer dereferencing	✓	✓			
80. floating-point overflow		✓			
81. floating-point overflow					8
82. arithmetic overflow	✓			✓	
83. arithmetic overflow	✓	✓		✓	
84. pointer dereferencing	✓	✓		✓	
85. pointer dereferencing	✓	✓		✓	
86. pointer dereferencing	✓	✓			
87. pointer dereferencing	✓	✓			
88. floating-point overflow					7
89. pointer dereferencing	✓	✓		✓	
90. pointer dereferencing	✓	✓		✓	
91. floating-point overflow	✓	✓			
92. arithmetic overflow	✓	✓		✓	
93. arithmetic overflow	✓	✓		✓	
94. pointer dereferencing	✓	✓			
95. pointer dereferencing	✓	✓		✓	
96. floating-point overflow					8
97. floating-point overflow			✓		
98. floating-point overflow					15
99. precondition	✓	✓		✓	
100. precondition	✓	✓			
101. variant decreases	✓		✓	✓	
102. variant decreases	✓	✓		✓	
103. pointer dereferencing	✓	✓		✓	
104. pointer dereferencing	✓	✓		✓	

	105. variant decreases	✓	✓		✓		
	106. variant decreases	✓	✓		✓		
Total for behavior and safety VCs						1,400	11 min 56
Total						9,084	34 min 21

```
/*@ axiomatic dirichlet_maths {
  @   logic real c;
  @   logic real p0(real x);
  @   logic real psol(real x, real t);
  @
  @   axiom c_pos: 0 < c;
  @
  @   logic real psol_1(real x, real t);
  @   axiom psol_1_def: \forall real x; \forall real t;
  @     \forall real eps; \exists real C; 0 < C && \forall real dx;
  @     0 < eps ==> \abs(dx) < C ==>
  @     \abs((psol(x + dx, t) - psol(x, t)) / dx - psol_1(x, t)) < eps;
  @
  @   logic real psol_11(real x, real t);
  @   axiom psol_11_def: \forall real x; \forall real t;
  @     \forall real eps; \exists real C; 0 < C && \forall real dx;
  @     0 < eps ==> \abs(dx) < C ==>
  @     \abs((psol_1(x + dx, t) - psol_1(x, t)) / dx - psol_11(x, t)) < eps;
  @
  @   logic real psol_2(real x, real t);
  @   axiom psol_2_def: \forall real x; \forall real t;
  @     \forall real eps; \exists real C; 0 < C && \forall real dt;
  @     0 < eps ==> \abs(dt) < C ==>
  @     \abs((psol(x, t + dt) - psol(x, t)) / dt - psol_2(x, t)) < eps;
  @
  @   logic real psol_22(real x, real t);
  @   axiom psol_22_def: \forall real x; \forall real t;
  @     \forall real eps; \exists real C; 0 < C && \forall real dt;
  @     0 < eps ==> \abs(dt) < C ==>
  @     \abs((psol_2(x, t + dt) - psol_2(x, t)) / dt - psol_22(x, t)) < eps;
  @
  @   axiom wave_eq_0: \forall real x; 0 <= x <= 1 ==> psol(x, 0) == p0(x);
  @   axiom wave_eq_1: \forall real x; 0 <= x <= 1 ==> psol_2(x, 0) == 0;
  @   axiom wave_eq_2: \forall real x; \forall real t;
  @     0 <= x <= 1 ==> psol_22(x, t) - c * c * psol_11(x, t) == 0;
  @
  @   axiom wave_eq_dirichlet_1: \forall real t; psol(0, t) == 0;
  @   axiom wave_eq_dirichlet_2: \forall real t; psol(1, t) == 0;
  @
  @   logic real psol_Taylor_3(real x, real t, real dx, real dt);
  @   logic real psol_Taylor_4(real x, real t, real dx, real dt);
  @
  @   logic real alpha_3; logic real C_3;
  @   logic real alpha_4; logic real C_4;
  @
  @   axiom psol_suff_regular_3:
  @     0 < alpha_3 && 0 < C_3 &&
  @     \forall real x; \forall real t; \forall real dx; \forall real dt;
  @     0 <= x <= 1 ==> \sqrt(dx * dx + dt * dt) <= alpha_3 ==>
  @     \abs(psol(x + dx, t + dt) - psol_Taylor_3(x, t, dx, dt)) <=
  @       C_3 * \abs(\pow(\sqrt(dx * dx + dt * dt), 3));
  @
  @
```

```
  @   axiom psol_suff_regular_4:
  @     0 < alpha_4 && 0 < C_4 &&
  @     \forall real x; \forall real t; \forall real dx; \forall real dt;
  @     0 <= x <= 1 ==> \sqrt(dx * dx + dt * dt) <= alpha_4 ==>
  @     \abs(psol(x + dx, t + dt) - psol_Taylor_4(x, t, dx, dt)) <=
  @       C_4 * \abs(\pow(\sqrt(dx * dx + dt * dt), 4));
  @
  @   axiom psol_le: \forall real x; \forall real t;
  @     0 <= x <= 1 ==> 0 <= t ==> \abs(psol(x, t)) <= 1;
  @
  @   logic real T_max;
  @   axiom T_max_pos: 0 < T_max;
  @
  @   logic real C_conv; logic real alpha_conv;
  @   lemma alpha_conv_pos: 0 < alpha_conv;
  @ } */


/*@ axiomatic dirichlet_prog {
  @
  @  predicate analytic_error{L}
  @    (double **p, integer ni, integer i, integer k, double a, double dt)
  @    reads p[..][..];
  @
  @  lemma analytic_error_le{L}:
  @    \forall double **p; \forall integer ni; \forall integer i;
  @    \forall integer nk; \forall integer k;
  @    \forall double a; \forall double dt;
  @      0 < ni ==> 0 <= i <= ni ==> 0 <= k ==> 0 < \exact(dt) ==>
  @      analytic_error(p, ni, i, k, a, dt) ==>
  @      \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv ==>
  @      k <= nk ==> nk <= 7598581 ==> nk * \exact(dt) <= T_max ==>
  @      \exact(dt) * ni * c <= 1 - 0x1.p-50 ==>
  @      \forall integer i1; \forall integer k1;
  @        0 <= i1 <= ni ==> 0 <= k1 < k ==>
  @          \abs(p[i1][k1]) <= 2;
  @
  @  predicate separated_matrix{L}(double **p, integer leni) =
  @    \forall integer i; \forall integer j;
  @    0 <= i < leni ==> 0 <= j < leni ==> i != j ==>
  @    \base_addr(p[i]) != \base_addr(p[j]);
  @
  @
  @ logic real sqr(real x) = x * x;
  @ logic real sqr_norm_dx_conv_err{L}
  @   (double **p, real dx, real dt, integer ni, integer i, integer k)
  @   reads p[..][..];
  @
  @ lemma sqr_norm_dx_conv_err_0{L}:
  @   \forall double **p; \forall real dx; \forall real dt;
  @   \forall integer ni; \forall integer k;
  @   sqr_norm_dx_conv_err(p, dx, dt, ni, 0, k) == 0;
  @
```

```
  @ lemma sqr_norm_dx_conv_err_succ{L}:
  @   \forall double **p; \forall real dx; \forall real dt;
  @   \forall integer ni; \forall integer i; \forall integer k;
  @   0 <= i ==>
  @   sqr_norm_dx_conv_err(p, dx, dt, ni, i + 1, k) ==
  @     sqr_norm_dx_conv_err(p, dx, dt, ni, i, k) +
  @     dx * sqr(psol(1. * i / ni, k * dt) - \exact(p[i][k]));
  @
  @ logic real norm_dx_conv_err{L}
  @   (double **p, real dt, integer ni, integer k) =
  @   \sqrt(sqr_norm_dx_conv_err(p, 1. / ni, dt, ni, ni, k));
  @ } */


/*@ requires leni >= 1 && lenj >= 1;
  @ ensures \valid_range(\result, 0, leni - 1) &&
  @   (\forall integer i; 0 <= i < leni ==>
  @    \valid_range(\result[i], 0, lenj - 1)) &&
  @   separated_matrix(\result, leni);
  @ */
double **array2d_alloc(int leni, int lenj);

/*@ requires (l != 0) && \round_error(x) <= 5./2*0x1.p-53;
  @ ensures \round_error(\result) <= 14 * 0x1.p-52 &&
  @   \exact(\result) == p0(\exact(x));
  @ */
double p_zero(double xs, double l, double x);



/*@ requires
  @   ni >= 2 && nk >= 2 && l != 0 &&
  @   dt > 0. && \exact(dt) > 0. &&
  @   \exact(v) == c && \exact(v) == v &&
  @   0x1.p-1000 <= \exact(dt) &&
  @   ni <= 2147483646 && nk <= 7598581 &&
  @   nk * \exact(dt) <= T_max &&
  @   \abs(\exact(dt) - dt) / dt <= 0x1.p-51 &&
  @   0x1.p-500 <= \exact(dt) * ni * c <= 1 - 0x1.p-50 &&
  @   \sqrt(1. / (ni * ni) + \exact(dt) * \exact(dt)) < alpha_conv;
  @
  @ ensures
  @   \forall integer i; \forall integer k;
  @   0 <= i <= ni ==> 0 <= k <= nk ==>
  @   \round_error(\result[i][k]) <= 78. / 2 * 0x1.p-52 * (k + 1) * (k + 2);
  @
  @ ensures
  @   \forall integer k; 0 <= k <= nk ==>
  @   norm_dx_conv_err(\result, \exact(dt), ni, k) <=
  @     C_conv * (1. / (ni * ni) + \exact(dt) * \exact(dt));
  @ */
double **forward_prop(int ni, int nk, double dt, double v,
                      double xs, double l) {
```

```
/* Output variable. */
double **p;

/* Local variables. */
int i, k;
double a1, a, dp, dx;

dx = 1./ni;
/*@ assert dx > 0. && dx <= 0.5 &&
  @   \abs(\exact(dx) - dx) / dx <= 0x1.p-53;
  @ */

a1 = dt/dx*v;
a = a1*a1;
/*@ assert 0 <= a <= 1 && 0 < \exact(a) <= 1 &&
  @   \round_error(a) <= 0x1.p-49;
  @ */

p = array2d_alloc(ni+1, nk+1);

/* First initial condition and boundary conditions. */
/* Left boundary. */
p[0][0] = 0.;

/* Time iteration -1 = space loop. */
/*@ loop invariant 1 <= i <= ni &&
  @   analytic_error(p, ni, i - 1, 0, a, dt);
  @ loop variant ni - i; */
for (i=1; i<ni; i++) {
  p[i][0] = p_zero(xs, l, i*dx);
}
/* Right boundary. */
p[ni][0] = 0.;
/*@ assert analytic_error(p, ni, ni, 0, a, dt); */

/* Second initial condition (with p_one=0) and boundary conditions. */
/* Left boundary. */
p[0][1] = 0.;
/* Time iteration 0 = space loop. */
/*@ loop invariant 1 <= i <= ni &&
  @   analytic_error(p, ni, i - 1, 1, a, dt);
  @ loop variant ni - i; */
for (i=1; i<ni; i++) {
  /*@ assert \abs(p[i-1][0]) <= 2; */
  /*@ assert \abs(p[i][0]) <= 2; */
  /*@ assert \abs(p[i+1][0]) <= 2; */
  dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
  p[i][1] = p[i][0] + 0.5*a*dp;
}
/* Right boundary. */
p[ni][1] = 0.;
/*@ assert analytic_error(p, ni, ni, 1, a, dt); */
```

```
  /* Evolution problem and boundary conditions. */
  /* Propagation = time loop. */
  /*@ loop invariant 1 <= k <= nk &&
    @   analytic_error(p, ni, ni, k, a, dt);
    @ loop variant nk - k; */
  for (k=1; k<nk; k++) {
    /* Left boundary. */
    p[0][k+1] = 0.;
    /* Time iteration k = space loop. */
    /*@ loop invariant 1 <= i <= ni &&
      @   analytic_error(p, ni, i - 1, k + 1, a, dt);
      @ loop variant ni - i; */
    for (i=1; i<ni; i++) {
      /*@ assert \abs(p[i-1][k]) <= 2; */
      /*@ assert \abs(p[i][k]) <= 2; */
      /*@ assert \abs(p[i+1][k]) <= 2; */
      /*@ assert \abs(p[i][k-1]) <= 2; */
      dp = p[i+1][k] - 2.*p[i][k] + p[i-1][k];
      p[i][k+1] = 2.*p[i][k] - p[i][k-1] + a*dp;
    }
    /* Right boundary. */
    p[ni][k+1] = 0.;
    /*@ assert analytic_error(p, ni, ni, k + 1, a, dt); */
  }
  return p;
}
```

4. Because Compilation Does Matter

All the previous programs were formally proved correct with respect to their specifications. It should mean that they behave as formally described by the annotations. Unfortunately, this is not the case due to compilation discrepancies. The same program may give several answers on several environments.

This is due to architecture-dependent features. The first one is the x87 floating-point unit that uses a 80-bit internal floating-point registers on many Intel platforms. The second one is the fused multiply-add (FMA) instruction, supported for instance by the PowerPC and the Intel Itanium architectures, that computes $xy \pm z$ with a single rounding.

Then, the C standard states that "the values of operations with floating operands are evaluated to a format whose range and precision may be greater than required by the type". This optimization opportunity applies to the use of a FMA operator for computing the expression $a \times b + c$, as the intermediate product is then performed with a much greater precision. This also means the use of 80-bit registers and computations are allowed at will. In particular, for each operation, the compiler may choose to round the infinitely-precise result either to extended precision, or to double precision, or first to extended and then to double precision. The latter is called a double rounding.

Consider this instance of the Fast-Two-Sum [Dek71] that computes exactly the rounding error of a FP addition.

```
int main () {
  double y, z;
  y = 0x1p-53 + 0x1p-78;       // y = 2^-53 + 2^-78
  z = 1. + y - 1. - y;
  printf("%a\n", z);
  return 1;
}
```

This very simple program compiled with GCC 4.6.3 gives three different answers on an x86 architecture depending on the instruction set and the chosen level of optimization.

Compilation options	Program result
`-O0` (x86-32)	`-0x1p-78`
`-O0` (x86-64)	`0x1.ffffffp-54`
`-O1`, `-O2`, `-O3`	`0x1.ffffffp-54`
`-Ofast`	`0x0p+0`

How can we explain the various results? In all cases, y is computed exactly: $y = 2^{-53} + 2^{-78}$. With the "-O0" optimization for the 32-bit instruction set, all the computations are performed with extended precision and rounded in double precision only once at the end. With the "-O0" optimization for the 64-bit instruction set, all the computations are performed with double precision. With "-O1" and higher, the intermediate value $(1 + y) - 1$ is pre-computed by the compiler as if performed with double precision; the program effectively computes only

the last subtraction and the result does not depend on the instruction set. With "-Ofast", there is no computation at all in the program but only the output of the constant 0. This optimization level turns on `-funsafe-math-optimizations` which allows the reordering of FP operations. It is explicitly stated in GCC documentation that this option "can result in incorrect output for programs which depend on an exact implementation of IEEE or ISO rules/specifications for math functions".

Many more examples of strange FP behaviors have been listed by Monniaux [Mon08]. These behaviors may break the fine-tuned proofs done before. Several possibilities to overcome this are presented in this chapter.

A first choice is the direct analysis of the assembly code [Yu92, NM11]. Instead of the C code, we compile it and consider the assembly code where all architecture-dependent information is known, such as the precision of each operation. We are left to check that the specifications, written at the C level, are valid with respect to the assembly code generated by the compiler. There is therefore no overestimation of the error as each precision used is known. A drawback is that tools were not always able to interpret all the compiler optimizations, such as inlining.

4.1 Covering All Cases

Another approach I worked on with Nguyen is to cover all cases: the verification will guarantee that the specification will hold whatever the hardware, the compiler and the optimization level. This is especially useful when the program will run on several platforms, such as different planes in the KB3D example of Section 3.7.

4.1.1 Double Rounding

First, we will handle all the possible uses of extended registers and therefore all possible double roundings. The following theorem has been formally proved in Coq:

Theorem 4.1.1 *For a real number x, let $\square(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$. Then, we have either*

$$\left(|x| \geq 2^{-1022} \text{ and } \left|\frac{x-\square(x)}{x}\right| \leq 2050 \times 2^{-64}\right)$$

or

$$\left(|x| \leq 2^{-1022} \text{ and } |x-\square(x)| \leq 2049 \times 2^{-1086}\right).$$

This theorem is the basis of our approach to correctly prove numerical programs whatever the hardware. These bounds are tight as they are reached in all cases where $\square$ is the double rounding. They are a little bigger than the ones for 64-bit rounding (2050 and 2049 instead of 2048) for both cases. These bounds are therefore both correct, very tight, and just above the 64-bit's. The proof is summarized in Figure 4.1: the various cases for $\square$ and intervals for x are successively studied and the corresponding errors are bounded.

In practice, we will use:

$$|x-\square(x)| \leq \varepsilon_{\text{DR}} \times |x| + \eta_{\text{DR}}$$

with $\varepsilon_{\text{DR}} = 2050 \times 2^{-64}$ and $\eta_{\text{DR}} = 2049 \times 2^{-1086}$.

In strict IEEE-754, where inputs and outputs are on 64 bits, we can set $\eta = 0$ for addition and subtraction. Unfortunately here, inputs may be 80-bit numbers so η cannot be set to 0.

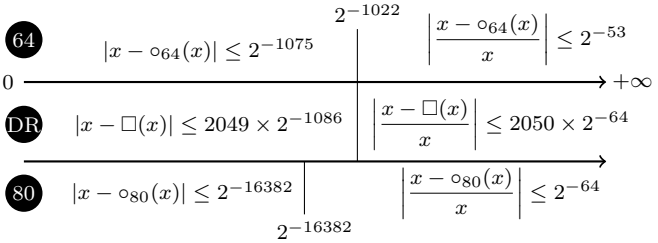

Figure 4.1: Rounding error in 64-bit, 80-bit, and in double rounding (DR). The latter corresponds to Theorem 4.1.1

.

4.1.2 FMA

Theorem 4.1.1 gives rounding error formulas for various roundings denoted by $\square$ (64-bit, 80-bit and double rounding). Now, we consider the FMA that computes $x \times y \pm z$ with one single rounding. To consider the cases whether a FMA is used, the idea is very simple: we consider a FMA as a *rounded* multiplication followed by a rounded addition. And we only have to consider another possible "rounding" that is the identity: $\square(x) = x$. This specific "rounding" magically covers all the FMA possibilities: the result of a FMA is $\square_1(x \times y + z)$, that may be considered as $\square_1(\square_2(x \times y) + z)$ with $\square_2$ being the identity. So we handle in the same way all operations even in presence of FMA or not, by considering one rounding for each basic operation (addition, multiplication, and so on). Of course, the formulas of Theorem 4.1.1 easily hold for the identity rounding.

What is the use of this strange rounding? The idea is that each *basic* operation (addition, subtraction, multiplication, division, square root, negation and absolute value) will be considered as rounded with a $\square$ that may be one of the four possible roundings ($\circ_{64}(x)$, $\circ_{80}(x)$, $\circ_{64}(\circ_{80}(x))$, x). Some impossible cases are allowed: for example, all computations being exact. The important point is that *all* the actual possibilities are *included* in all the considered possibilities. And all of them have a rounding error bounded by Theorem 4.1.1.

4.1.3 How Can This be Applied?

This needs to be applied in practice to actual C programs. For that, we use a pragma that tells Frama-C that the compiler may use extended registers and FMA. That pragma changes the definition of the floating-point operations. More precisely, we change the operation postcondition, that is to say how an operation result is defined in the VCs.

For all basic operations: $x = a + b$, $a - b$, $a \times b$, a/b, $\sqrt{a}$, we modify the postcondition as:

$$\begin{array}{c}\text{strict IEEE-754 standard}\\ \text{only one possible result: } f = \circ_{64}(x)\\ \downarrow\\ \text{FMA and extended registers}\\ f \text{ is any real such that } |f - x| \leq \varepsilon_{\text{DR}}|x| + \eta_{\text{DR}}\end{array}$$

with $\varepsilon_{\text{DR}} = 2050 \times 2^{-64}$ and $\eta_{\text{DR}} = 2049 \times 2^{-1086}$.

As seen in the KB3D example of Section 3.7, this works in practice. Using Gappa, error bounds can be proved, which are valid whatever the platform and the compilation choices.

4.1.4 Associativity for the Addition

To push the preceding idea even further, we may also take into account that the compiler may reorganize additions. The idea here is that we will change the rounding error formula for the addition in order to guarantee that, even if $(a+b)+c$ is transformed into $a+(b+c)$ by the compiler, what is proved will still hold.

For that, we use a formula of this form: $|a \oplus b - (a+b)| \leq \varepsilon' \cdot (|a|+|b|) + \eta'$ (with given ε' and η'). Instead of an error proportional to $|a+b|$ as before, that is about the final result, the error is proportional to $|a|+|b|$. This is a huge difference that handles the cancellations, but may increase the proved bounds on the rounding error.

To guarantee this approach, we proved the following theorem. Given a ε, we set $\varepsilon_n = (1+\varepsilon)^n - 1$.

Theorem 4.1.2 *Assume an integer n such that $n \leq \frac{1}{\varepsilon}$, a sequence of real numbers $(a_i)_{0 \leq i \leq n}$ and a real I,*

We assume that, if we set the addition postcondition as: $x \oplus y$ is any real number such that
$$|x \oplus y - (x+y)| \leq \varepsilon_n \cdot (|x|+|y|) + n \cdot \eta,$$
we are able to deduce that $|S_n^{\sigma_1} - \sum_0^n a_i| \leq I$ for an ordering σ_1 of the additions.

Now we set the addition postcondition as: $x \oplus y$ is any real number such that
$$|x \oplus y - (x+y)| \leq \varepsilon \cdot |x+y| + \eta.$$
Then, whatever the ordering σ_2 of the additions, we have $|S_n^{\sigma_2} - \sum_0^n a_i| \leq I$.

The proof has been published [BN11]. Its meaning is that, if we are able to prove a bound on the rounding error for a sum in a program using our loose postconditions (meaning $|x \oplus y - (x+y)| \leq \varepsilon_n \cdot (|x|+|y|) + n \cdot \eta$), then this bound is still correct whatever the compiler reorganization. The idea is: what is proved using Frama-C and the loose postconditions still holds with another ordering. The ε will be the ε_{DR} of the preceding Section.

How tight is the chosen postcondition? Let us discard the ε^2 terms and consider $n+1$ values: $a_0 = 1$ and $a_i = 2^{-53}$. Then the left-most parenthesing gives a_0 and the right-most gives $1 + n2^{-53}$, that is to say a difference of $n\varepsilon \approx \varepsilon_n$. This example shows that our bound is as tight as possible.

The question left is the choice of n. A solution is to look into the program to have an overestimation of n. We did not put this idea in practice and decided that, for our examples, 16 will be enough. In this case, for we have the same postcondition as before for multiplication, division and square root. For addition and subtraction, we set the postcondition as:

strict IEEE-754 standard
only one possible result: $a \oplus b = \circ_{64}(a+b)$
$\downarrow$
FMA and extended registers
$a \oplus b$ is any real such that $|a \oplus b - (a+b)| \leq \varepsilon|a+b| + \eta$
$\downarrow$
FMA, extended registers and addition reorganization
$a \oplus b$ is any real such that $|a \oplus b - (a+b)| \leq \varepsilon'(|a|+|b|) + \eta'$

For $n = 16$, we choose $\varepsilon' = 2051 \cdot 2^{-60}$ so that $\varepsilon' \geq \varepsilon_{16}^{\text{DR}}$ and $\eta' = 16\eta_{\text{DR}} = 2049 \times 2^{-1082}$.

Contrary to the previous method with only 80-bit computations and FMA, the obtained results taking associativity into account are not convincing. Even if the bounds are as tight as possible in the general case, they eventually give very coarse results, meaning large error bounds. Moreover, the assumption bounding the number of reorganized additions ($n \leq 16$) is very strong.

4.2 Correct Compilation

Rather than trying to account for all the changes a compiler may have silently introduced in a FP program, Jourdan, Leroy, Melquiond, and I have focused on getting a correct and predictable compiler that supports FP arithmetic. To build our compiler, we started from CompCert [Ler09], a compiler formally verified in Coq and we extended it with FP arithmetic. CompCert comes with a mathematical proof of semantic preservation that rules out all possibilities of miscompilation. It assumes that S is a source C program free of undefined behaviors. It further assumes that the CompCert compiler, invoked on S, does not report a compile-time error, but instead produces executable code E. Then, the semantic preservation states that any observable behavior B of E is one of the possible observable behaviors of S. Intuitively, the semantic preservation theorem says that the executable code produced by the compiler always executes as prescribed by the semantics of the source program.

This leaves two important degrees of freedom to the compiler. First, a C program can have several legal behaviors, as in the use of extended registers, and the compiler is allowed to pick any one of them. Second, undefined C behaviors need not be preserved during compilation, as the compiler can optimize them away.

Concerning arithmetic operations in C and in assembly languages, their semantics are specified in terms of two Coq libraries, Int and Float, which provide Coq types for integer and FP values, and Coq functions for the basic arithmetic and logical operations, for conversions between these types, and for comparisons. In contrast, in early versions of CompCert, the Float library was not constructed, but only axiomatized: the type of FP numbers is an abstract type, the arithmetic operations are just declared as functions but not realized, and the algebraic identities exploited during code generation are not proved to be true, but only asserted as axioms. Consequently, conformance to IEEE-754 could not be guaranteed, and the validity of the axioms could not be machine-checked. Moreover, this introduced a regrettable dependency on the host platform (the platform that runs the CompCert compiler), as numerical computations at compile-time are mapped to FP operations provided by OCaml. However, OCaml's FP arithmetic is not guaranteed to implement IEEE-754 double precision: on the x86 architecture running in 32-bit mode, OCaml compiles FP operations to x87 machine instructions, resulting in double-rounding issues.

CompCert is now based on Flocq that constructs FP numbers; CompCert therefore correctly computes constant propagation and evaluation of FP literals [BJLM13]. This development has been integrated into CompCert since version 1.12. It also allows to formally verify algebraic simplifications over floating-point operations (such as $x \oslash 2 \to x \otimes 2^{-1}$) and formally verify code generation for conversions between FP and integer numbers (in the two directions), that are synthesized in software for hardware platforms that do not natively support them.

Some of these conversions are using properties based on the rounding to odd, that was first used by Goldberg [Gol91] without naming it and then generalized by Melquiond and me [BM08]. The informal definition of odd rounding is the following: when a real number

is not representable, it will be rounded to the adjacent FP number with an odd integer significand. Let us have two different reasonable formats, characterized by φ and φ_e. We assume that $\forall e \in \mathbb{Z}, \quad \varphi_e(e) \leq \varphi(e) - 2$. This informally means that we have an extended format with at least two more digits. We assume that the radix is even. Then, if $\square_e^{odd}$ denotes the rounding to odd in format φ_e, and if $\circ$ denotes a rounding to nearest in format φ, with an arbitrary rule for ties, we have $\forall x \in \mathbb{R}, \quad \circ\left(\square_e^{odd}(x)\right) = \circ(x)$. Examples of use include several conversions from 64-bit integers to FP numbers [BJLM14].

This approach gives a correct and predictable compiler that conforms to the IEEE-754 standard. This means that, among the several possibilities allowed by the ISO C standard, we have chosen a single way to compile and we have formally proved its correctness. This compilation choice can be discussed: for example, all intermediate results are computed in double precision, therefore with (usually) less accuracy than with extended registers. This therefore agrees with our deductive verification assumptions. Another reason is to favor reproducibility over possible higher accuracy.

A conclusion is that these tools fit together. More precisely, given a C program verified using Frama-C/Jessie/Why and compiled using CompCert, then the executable code respects its specifications.

5. Conclusion and Perspectives

5.1 Conclusion

5.1.1 What is Not Here

Science is not a straight course and several other topics have interested me at some point without resulting in a proved program. This mainly concerns properties and algorithms formally proved in Coq using PFF. This includes two's complement formats [BD02, BD04a, Bol06a], computation of predecessor and successor [RZBM09], argument reduction [LBD03, BDL09], faithful polynomial evaluation [BD04b]. This also includes the existence of correcting terms for the common operations [BD03], but also for the FMA [BM05, BM11b] and how to compute them.

A last theme is expansions: a value is represented by the unevaluated sum of several FP numbers as advocated by Dekker and Priest at the beginning of the 70s [Dek71, Pri91] and popularized later [She97, ORO05]. Some algorithms have been proved [BD01], but this has not been pushed forward with a program proof.

This is not the place to describe my popular science activities. Nevertheless, I believe that displaying science and especially computer science to kids is both a necessary and pleasant task. Kids will turn into adults that should understand the digital world, more than only use it. Moreover, it is a good way to turn them into scientists (especially girls). A study has shown that encouragement and exposure are key controllable factors for whether or not young women decide to pursue a Computer Science degree [Goo14].

5.1.2 What is Here

We have shown that programs using floating-point arithmetic can be proved with a very high level of guarantee. We have seen many limits, including inherent ones due to the environment: hardware, compiler. Here are others due to the available power:

- machine power: for the Dirichlet example, a Coq theorem (just stating the theorem to prove) is more than 500 lines long and consists in about 350 hypotheses. This is indeed pushing the limits of Coq and its graphical interface. The `intuition` tactic then takes 150 seconds on a 3 Ghz machine.

- human power: hundreds of lines of tactics are left for the user to write. Even if the `gappa` tactic has improved the situation, there is also room for other automated tactics to solve typical goals, such as $\beta^{9-9p}+9\beta^{8-8p}+36\beta^{7-7p}+84\beta^{6-6p}+126\beta^{5-5p}+126\beta^{4-4p}+84\beta^{3-3p}+36\beta^{2-2p}+9\beta^{1-p} \leq 37\beta^{2-2p}+9\beta^{1-p}$ for typical radix β (integer greater than 1) and precision p (find the least one greater than one that fulfills this).

This overview of tricky programs has also shown us that proofs in the usual case (no underflow, no overflow) are not more complicated than clear precise proofs on paper. This shows

that the underlying libraries are usually sufficient. There are two problems left. Overflow here is handled by putting preconditions so that Gappa proves all the overflow VCs. This method can be discussed: it is sometimes not optimal, but it works very well with satisfactory results. Underflow is not as easy to dismiss: it may create large relative errors that have to be handled in one way or another. Sometimes, underflow is addressed as the normal case, for example in Sterbenz (page 24) as an underflowing result of an addition is correct. This is also the case when we only consider the absolute error, as in the numerical analysis examples (page 38 and page 40). Sometimes, underflow has to be prevented using preconditions. It may be hypotheses on the inputs, as for Dekker (page 28) or for the accurate discriminant (page 30) or on the output as for the area of a triangle (page 32). A general way to handle underflow should be looked for.

Another point is what formal methods bring along. More than just a verification of proofs, they are useful for the modification of proofs. It is easy to state that "this result can easily be generalized to any radix", but it is very cumbersome, and error-prone to check a pen-and-paper proof to see where it fails for another radix than 2. Formal proofs are easy to work on afterwards. Just take your proof and modify your hypotheses, you are left to debug the proof as it will most certainly fail. But it is easy to see where and why it fails, and therefore if and how it can be corrected. This is a formidable tools for making generic proofs, and as generic as possible. While intuition gives the first proof of a result, formal methods help to polish it in order to get less or tighter hypotheses.

This outcome of proved programs and interactive demonstrations is 9 programs, 276 lines of annotations for 122 lines of C, and more than 16,000 dedicated lines of Coq. The supporting libraries, done with several co-authors, amount to more than 100,000 lines of Coq. These investigations, especially about numerical analysis, have shown us several research directions, about the tools and the application domains.

5.2 Perspectives About Tools

In theory, the user should not adapt to his/her tools but adapt his/her tools to his/her needs.

5.2.1 Many Tools, Many Versions

I have been using many tools and versions over the years. I began using the PFF library in 2001 and it was using Coq 6.3. I then have been through all versions of Coq. Each upgrade of Coq came with various incompatibilities, mostly minor, but cumbersome and time-consuming. In particular, the upgrade from 6.* to 7.0 came with a major language change and a large modification of the theorem names of the standard library for real numbers.

Concerning the other software described in Section 2.1, the first proofs of programs were done in 2006 using Caduceus. Later, Caduceus was subsumed by the Frama-C platform, with the Jessie plugin and the Why 2.* platform. Most of the described verified programs were done in that setting, still using PFF. The next Why platform Why3, was developed from 2011 and a few proofs now use it, with Frama-C, the Jessie plugin, and Flocq.

There is a persistent question: are the guarantees changing with the used version of the tools? Some programs are proved one way, but not another (typically with Why2 and not Why3). Does this change the trust of these programs? How will this trust evolve if the tools are not maintained? Reproducibility is an important factor of trust. If proofs or experiments cannot be re-run, they lose some value, and this tool evolution decreases the obtained trust.

I am not saying tools should not evolve. This is necessary, and especially for research tools: sometimes, you have to throw everything away and start anew.

Unfortunately, I cannot see any solution. Backward compatibility is too much to ask for research tools and would severely restrict them. Maintaining a gallery of proved programs from one version to another is a heavy task I consider useful. It permits one to test the software evolutions and to stimulate the fact that at least the same things should be doable.

A last perspective about the tools is of course to make them talk together. Abstract interpretation is automatic and its domains are powerful while deductive verification is more protean. For instance, Fluctuat is able to prove assertions that Gappa is not so that they would have to be proved in Coq in our framework. Similarly, Fluctuat could benefit from assertions such as $p[i][k] \leq 2$ obtained by mathematical proofs. The mix of static analyses is not a new idea, but I think it is worth the time spent.

5.2.2 Coq Libraries

The Coq libraries Flocq for floating-point arithmetic (see Section 2.3.2) and Coquelicot (see Section 2.4) should be maintained and increased when pertinent. A recent work by P. Roux has been partially integrated into Flocq [Rou14].

Coquelicot is currently being generalized for more generic spaces. To easily handle sets other than reals, such as complex numbers and matrices, we have developed an algebraic hierarchy to take advantage of our previous work while preventing the duplication of theorems. Other extensions include definitions and proofs of elementary functions using complex numbers. It would be feasible in a real-only setting, but it would be much more straightforward using complex numbers. Another extension concerns the improvement of automated methods, so as to handle predicates such as integrability and multivariate continuity. Other extensions are required by computer algebra applications in order to work formally and comfortably with differential equations, Padé approximants, generalized Fourier series, or Lyapunov certificates. The goal is to obtain comprehensive volumes of formalized mathematics, for the benefit of a much larger community of users of proof assistants.

A more short-term perspective is the end of the PFF library (see Section 2.3.1). Like the standard library for real numbers, it is now an old library that has not evolved much. Results have been added over the years, but a major change was needed. We therefore developed the Flocq library. Now is the time to take the best part of PFF into Flocq. This means some translations, probably by equivalence theorems, so that both the biggest and the keenest results are not lost.

5.2.3 Exceptional Behaviors

Another interesting topic is that of exceptional behaviors. A good survey is available [Hau96] that advocates that exceptions must be available to the programmer in the programming languages. Such primitives should then be interpreted by the tools. This is not straightforward as exception flags have a complex and sometimes processor-dependent behavior.

Another way to detect exceptional behavior is a static analysis based on symbolic execution that provides the user with inputs that create exceptional behavior [BVLS13]. They seem to detect a large number of real runtime exception, even if they cannot detect all of them.

5.3 Perspectives About Programs

Tools should be enhanced, but tools should also be exercised. This means that formal verification of programs should be kept on, with chosen application domains.

5.3.1 Computer Arithmetic

The first application domain is of course computer arithmetic. My goal is to go from well-known facts to formally proved facts. As many communities in computer science, the floating-point arithmetic community has a strong background, even if recent. It is based on several articles/books/people that serve as reference to new results. "It is well known that..." is ubiquitous and true! This knowledge is known by the community, shared and taught (even if in the precise case of computer arithmetic, I believe it is not taught enough). The idea is here to formalize and prove all this common knowledge to create a formally-verified base to build upon. My goal is to write an article similar to Goldberg's "What every computer scientist should know about floating point arithmetic" [Gol91] which would be both up-to-date and formally verified. The updating is necessary as the IEEE-754 standard is now ubiquitous (it is even in GPUs) and many features such as FMA have to be described.

Another natural perspective is the generalization of these well-known algorithms to radix 10. Decimal floating-point units are available and we would like to use common algorithms on these units. Some algorithms fail in radix 10, such as Fast-Two-Sum [Knu97], but most apply. This should be formally verified, as such generalization are error-prone.

5.3.2 Multi-Precision Libraries

Another application domain is that of multi-precision libraries, as many users run them, sometimes without even knowing it (while using a compiler for instance). Two very different approaches are available for getting extra precision.

Expansions, as also cited above in Section 5.1.1 are a way to get extended precision. The idea is to keep several FP numbers. Their sum represents a value with more precision than a single FP number. Algorithms for basic operations on expansions have been proved [BD01], but not programs. A more interesting perspective is to consider limited expansions, meaning 2 FP numbers, usually called double double or 4 FP numbers, usually called quad double. They are used to get additional precision but with a small extra cost, as the very fast FP operations are still used. Available libraries are efficient [HLB01, BHLT02], but their specification is quite weak. How many digits exactly are correct and how much overlap is allowed could be precisely specified and proved. In particular, a difficult point will be overflow. It would require an additional study of basic blocks such as Two-Sum or Fast-Two-Sum to get a precondition as tight as possible. Other applications such as accurate dot product [ORO05] with similar basic blocks could then be looked into.

The most common way to get multiple-precision arithmetic is to have a huge mantissa and an exponent. It removes the limits of expansions due to underflow and overflow, but may require more memory. As for speed, libraries such as MPFR have benefited from a large algorithmic effort that has made them efficient. MPFR [FHL+07] is a challenge for the verification of programs for two reasons, beside its use in GCC. First, the algorithms are complex, but well-documented in [MPF14]. Next, the programs themselves are complex as they reuse a lot the memory for efficiency. A first try was made by Bertot, Magaud and

Zimmermann [BMZ02], but only for the integer square root. The program works in-place so there was already work on memory management, and this was done using a precursor of Caduceus. As for memory management and integer computation, a recent very nice work is due to Affeldt [Aff13]: his goal is to formally prove low-level arithmetic functions and cryptography primitives. The methodology relating Coq proofs and assembly code is very interesting and could probably be applied.

5.3.3 Computational Geometry

Another domain where formal verification is called for is computational geometry. Many such algorithms are unstable, such as convex hull computations and Delaunay triangulations. One reason is that they use numerical computations to make boolean decisions. Typical examples are orientation predicates: inputs are numeric values while the answer is right or left. Then, a wrong answer is just plain wrong. Stability is therefore hard to achieve and calls for a high guarantee. A large library of computational geometry is CGAL [The14], that has always taken great care in the chosen numbers and computations [HHK+00]. For example, it uses semi-static floating-point filters: the idea is to first compute in basic FP numbers ("quick and dirty"), then make a test called a filter. If the filter succeeds, then the obtained result is sure to be correct. If the filter fails, call a slower multi-precision routine that will always gives the correct answer. It has the advantage of being correct in all cases and quick in most cases. Of course, it heavily relies on the filter that decides the correctness. The static filter for the orientation-2 predicate has been previously formally proved [MP07]. A larger-scale verification of the whole library could be envisioned: a full proof would require proofs about memory management, FP filters, but also about the multi-precision computations in case the FP computation was not accurate enough. It would tie together parts of the previous perspectives and therefore would be a great validating experiment, more than its intrinsic value.

5.3.4 Numerical Analysis

The F∮st project I led has shown us the initial gap between numerical analysis and formal certification. We have been pioneers in this topic and the formal proof community now takes an interest in this kind of problems. But their chosen approach is utterly different from ours. The chosen path is to embed the numerical scheme in the proof assistant [IH12, MS13, Imm14]. The method error is proved in the proof assistant and computations are done inside the proof assistant using constructive reals in Coq [MS13] and arbitrary-precision floating-point numbers in Isabelle/HOL [IH12, Imm14]. This is also explained in Figure 5.3.4: numerical analysis is embedded into the formal proof assistant.

This does prove the convergence of the numerical scheme and guarantees correct answers, but it seems to me definitely insufficient for the following reasons:

- This result is hard to accept and to understand for mathematicians working on numerical analysis. This is a language they do not understand.
- There is a gap between the program and the formalization of the scheme in the proof assistant. To run their scheme and to integrate them with other tools, numerical analysts program in Fortran or C/C++. Running the scheme into the proof assistant means no integration and a heavy slowdown that may not be acceptable for real-life applications.

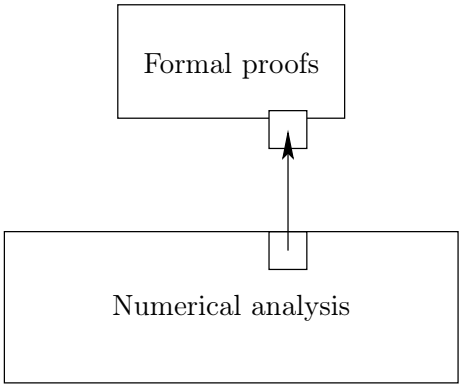

Figure 5.1: A common way to see the interaction between numerical analysis and formal methods: numerical analysis is embed into the proof assistant.

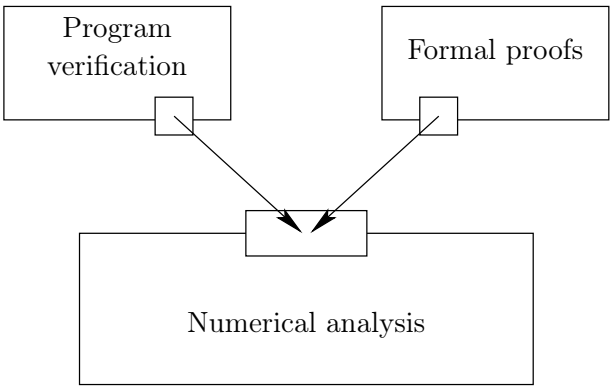

Figure 5.2: My point of view on the interaction between numerical analysis and formal methods: formal methods are to be applied to numerical analysis programs.

- Constructive reals and arbitrary-precision floating-point arithmetic allow one to guarantee the correctness of an effective algorithm. Nevertheless, real schemes are implemented using floating-point numbers because they are both fast and available from programming languages. Exact reals are easier to manage proofs, but we have shown in our case that `binary64` is sufficient for a good final accuracy, even if the proof of this fact was over-elaborate compared to what we expected.

My choice for formalizing numerical analysis is explained in Figure 5.3.4: instead of taking problems from numerical analysis and solving them inside proof assistants, we apply our methods and proofs to numerical analysis. We chose to start from a C program, specify its expected behavior, write the corresponding annotations and prove them. In both cases, the numerical scheme must be formalized and proved. In our case, we moreover prove that the C program corresponds to it. In the other case, the floating-point aspect is either ignored, or simplified by the use of multi-precision when needed. To be sure a given precision is enough is more complicated: we choose to guarantee the effective program and we do not have any sideways to ease our precision problems. To handle the C program also means to deal with a memory model and the updates of this memory. This is known to be a difficult problem, especially here as the computed values are put in a matrix (with the same base pointer) that is updated at each step.

5.3.5 Stability *vs* Stability

There are several meanings of the word stability. In computer arithmetic, an algorithm is stable if, when given slightly different inputs, it provides slightly different answers. The ratio between the input difference and the output difference gives the condition number [Hig02] that characterizes the stability of the function to be computed. For a function f, we set $y = f(x)$ and $y + \Delta y = f(x + \Delta x)$ as shown in Figure 5.3. In other words, an algorithm f is stable if a small Δx implies a small Δy. And a function f is backward-stable if a small Δy implies a small Δx. Stability provides a distance between f and its FP counterpart $\tilde{f}$, as we set in practice $y + \Delta y = \tilde{f}(x)$.

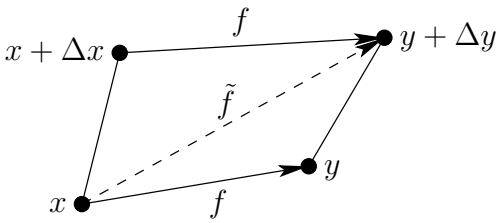

Figure 5.3: Forward and backward errors.

In numerical analysis, stability has another meaning. For ODE, it means the stability of the dynamic system, and often Lyapunov stability. For PDE, it means that the values do not diverge. This typically means that, given initial values (such as the values and derivative(s) at time 0), the values of the numerical scheme do not blow up.

Numerical analysts believe (as a rule of thumb) that stable schemes are numerically stable. This is a belief funded by experiments and examples. My opinion is that this fact is true and that it can and should be proved. This is a difficult task as it requires us to formalize a numerical scheme, its properties and to deduce its numerical stability. This result must be very generic to be applied to a large range of schemes, but also precise enough so that first it can be proved and second, the bound is good enough to guarantee a reasonable accuracy. For example, a bound such as $O(2^{-53})$ is useless.

5.3.6 Hybrid Systems

A last application domain tightly related with the previous numerical analysis is the study of hybrid systems. The main problem is how to formalize and to represent these hybrid systems in order to verify them. The most common method is to use high level models like hybrid automata. They are generally used to prove the reachability of some state [Fre08, ACH$^+$95]. A drawback is that the distance is very large between these automata, often required to be linear, and real-life programs. A solution is to analyze at the Simulink level [CM09]. Another solution is the KeYmaera tool [PQ08]. It uses a combination of automated theorem proving, real quantifier elimination, and symbolic computations in computer algebra systems. It has successfully verified case studies from train control, car control, and air traffic control.

Another way to fill the gap between the hybrid system and its code is the PhD thesis of Bouissou in French [Bou08] and in several articles [Bou09, BGP$^+$09]. The method is to extend C programs with ordinary differential equations presented as a C++ function. This allows to check any property of interest as the HybridFluctuat tool will automatically derives invariants on the whole system, FP properties included. An extension of our tools to handle

such an extension would be a great benefit. It would allow us to take advantage of each tool's strengths. In particular, deductive verification could prove some intricate mathematical part, while FP round-off errors would be handled automatically by Fluctuat's abstract domains.

Another interesting work is the formal definition of a semantics for hybrid systems [BBCP12]. Surprisingly, it is based on non-standard analysis for an easy manipulation of infinitesimal. This is not available in Coq, but is in ACL2(r) and Isabelle/HOL. Formalizing such a semantics in a proof assistant could give a common base to the numerous tools that deal with the verification of hybrid systems.

Bibliography

[AAHTH10] Behzad Akbarpour, Amr T. Abdel-Hamid, Sofiène Tahar, and John Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. *The Computer Journal*, 53(4):465–488, 2010.

[ABH+07] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H Schmitt. Verifying object-oriented programs with KeY: A tutorial. In *Formal Methods for Components and Objects*, pages 70–101. Springer, 2007.

[Abr05] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[AEF+05] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y Vardi, and Lenore D Zuck. Formal verification of backward compatibility of microcode. In *Computer Aided Verification*, pages 185–198. Springer, 2005.

[Aff13] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013.

[AM10] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, volume 6173 of *LNAI*, pages 127–141, Edinburgh, Scotland, July 2010. Springer.

[AS95] Mark D. Aagaard and Carl-Johan H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 7–10. IEEE Computer Society, 1995.

[Bar89] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[BBCP12] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3):877–910, 2012.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International journal on software tools for technology transfer*, 7(3):212–232, 2005.

[BCF+10] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal Proof of a Wave Equation Resolution Scheme: the Method Error. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the first Interactive Theorem Proving Conference (ITP)*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, July 2010. Springer.

[BCF+13] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013.

[BCF+14a] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.8*, 2014. `http://frama-c.cea.fr/acsl.html`.

[BCF+14b] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Trusting Computations: a Mechanized Proof from Partial Differential Equations to Actual Program. *Computers and Mathematics with Applications*, 68(3):325–352, 2014.

[BD01] Sylvie Boldo and Marc Daumas. A mechanically validated technique for extending the available precision. In *35th Asilomar Conference on Signals, Systems, and Computers*, pages 1299–1303, Pacific Grove, California, 2001.

[BD02] Sylvie Boldo and Marc Daumas. Properties of the subtraction valid for any floating point system. In *7th International Workshop on Formal Methods for Industrial Critical Systems*, pages 137–149, Málaga, Spain, 2002. Also in *Electronic Notes in Theoretical Computer Science*, volume 66(2). Elsevier, 2002.

[BD03] Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 79–86, Santiago de Compostela, Spain, 2003.

[BD04a] Sylvie Boldo and Marc Daumas. Properties of two's complement floating point notations. *International Journal on Software Tools for Technology Transfer*, 5(2-3):237–246, 2004.

[BD04b] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, 37(1-4):45–60, 2004.

[BDKM06] Sylvie Boldo, Marc Daumas, William Kahan, and Guillaume Melquiond. Proof and certification for an accurate discriminant. In *12th IMACS-GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Duisburg, Germany, September 2006.

[BDL09] Sylvie Boldo, Marc Daumas, and Ren-Cang Li. Formally Verified Argument Reduction with a Fused-Multiply-Add. *IEEE Transactions on Computers*, 58(8):1139–1145, 2009.

[BF07] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sarcedoti Coen, and Stephen M. Watt, editors, *16th Calculemus Symposium*, volume 5625 of *LNAI*, pages 59–74, Grand Bend, ON, Canada, 2009.

[BFM+13] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments (5th International Conference VSTTE)*, volume 8164 of *LNCS*, pages 191–201, Atherton, USA, May 2013. Springer.

[BGM06] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[BGP+09] Olivier Bouissou, Eric Goubault, Sylvie Putot, Karim Tekkal, and Franck Vedrine. HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 620–626, Berlin, Heidelberg, 2009. Springer-Verlag.

[BHH12] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI '12: Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[BHLT02] David H. Bailey, Yozo Hida, Xiaoye S. Li, and Brandon Thompson. ARPREC: An arbitrary precision computation package. Technical report, University of California, 2002.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[BJLM13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto

Nannarelli and Peter-Michael Seideland Ping Tak Peter Tang, editors, *Proceedings of the 21th IEEE Symposium on Computer Arithmetic*, pages 107–115, Austin, Texas, USA, April 2013.

[BJLM14] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 2014. `https://hal.inria.fr/hal-00862689`.

[BJMD+12] Nicolas Brisebarre, Mioara Joldeş, Érik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Paşca, Laurence Rideau, and Laurent Théry. Rigorous polynomial approximation using Taylor models in Coq. In *NASA Formal Methods*, pages 85–99. Springer, 2012.

[BLM12] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Improving Real Analysis in Coq: a User-Friendly Approach to Integrals and Derivatives. In Chris Hawblitzel and Dale Miller, editors, *Proceedings of the The Second International Conference on Certified Programs and Proofs*, volume 7679 of *LNCS*, pages 289–304, Kyoto, Japan, 2012.

[BLM14a] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 2014.

[BLM14b] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, 2014. `http://hal.inria.fr/hal-00806920`.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[BM98] Martin Berz and Kyoko Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4(4):361–369, 1998.

[BM05] Sylvie Boldo and Jean-Michel Muller. Some functions computable with a fused-mac. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the 17th Symposium on Computer Arithmetic*, pages 52–58, Cape Cod, USA, 2005.

[BM06] Sylvie Boldo and César Muñoz. Provably faithful evaluation of polynomials. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, volume 2, pages 1328–1332, Dijon, France, April 2006.

[BM08] Sylvie Boldo and Guillaume Melquiond. Emulation of FMA and Correctly-Rounded Sums: Proved Algorithms Using Rounding to Odd. *IEEE Transactions on Computers*, 57(4):462–471, 2008.

[BM11a] Sylvie Boldo and Guillaume Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.

[BM11b] Sylvie Boldo and Jean-Michel Muller. Exact and Approximated error of the FMA. *IEEE Transactions on Computers*, 60(2):157–164, February 2011.

[BMK06] Martin Berz, Kyoko Makino, and Youn-Kyung Kim. Long-term stability of the tevatron by verified global optimization. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 558(1):1–10, 2006.

[BMZ02] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3-4):225–252, 2002.

[BN10] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, number NASA/CP-2010-216215 in NASA Conference Publication, pages 14–23, Washington D.C., USA, April 2010.

[BN11] Sylvie Boldo and Thi Minh Tuyen Nguyen. Proofs of numerical programs when the compiler optimizes. *Innovations in Systems and Software Engineering*, 7:151–160, March 2011.

[Bol04a] Sylvie Boldo. Bridging the gap between formal specification and bit-level floating-point arithmetic. In Christiane Frougny, Vasco Brattka, and Norbert Müller, editors, *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 22–36, Schloss Dagstuhl, Germany, 2004.

[Bol04b] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, November 2004.

[Bol06a] Sylvie Boldo. Formal Proofs about DSPs. In Bruno Buchberger, Shin'ichi Oishi, Michael Plum, and Sigfried M. Rump, editors, *Algebraic and Numerical Algorithms and Computer-assisted Proofs*, number 05391 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany.

[Bol06b] Sylvie Boldo. Pitfalls of a full floating-point proof: Example on the formal proof of the Veltkamp/Dekker algorithms. In *Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR)*, pages 52–66, Seattle, USA, August 2006.

[Bol09] Sylvie Boldo. Floats & Ropes: a case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *LNCS - ARCoSS*, pages 91–102, Rhodos, Greece, July 2009. Springer.

[Bol13] Sylvie Boldo. How to compute the area of a triangle: a formal revisit. In Alberto Nannarelli and Peter-Michael Seideland Ping Tak Peter Tang, editors, *Proceedings of the 21th IEEE Symposium on Computer Arithmetic*, pages 91–98, Austin, Texas, USA, April 2013.

[Bou08] Olivier Bouissou. *Analyse statique par interpretation abstraite de systèmes hybrides*. PhD thesis, Ecole Polytechnique, 2008.

[Bou09] Olivier Bouissou. Proving the correctness of the implementation of a control-command algorithm. In Jens Palsberg and Zhendong Su, editors, *Static Analysis Symposium*, volume 5673 of *LNCS*, pages 102–119. Springer Berlin Heidelberg, 2009.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In *19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[BVLS13] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 549–560. ACM, 2013.

[Car95] Victor A. Carreño. Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical Report Technical Memorandum 110189, NASA Langley Research Center, 1995.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *LNCS*, pages 21–30. Springer Berlin Heidelberg, 2005.

[CCKL08] Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198-2 of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[CGRS14] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamarić, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 43–52. ACM, 2014.

[CGZ99] Edmund M. Clarke, Steven M. German, and Xudong Zhao. Verifying the SRT division algorithm using theorem proving techniques. *Formal Methods in System Design*, 14(1):7–44, 1999.

[CHJL11] Sylvain Chevillard, John Harrison, Mioara Joldeş, and Christoph Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523–1543, April 2011.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 233–247. Springer Berlin Heidelberg, 2012.

[CM95] Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *1995 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, 1995. supplemental proceedings.

[CM09] Alexandre Chapoutot and Matthieu Martel. Abstract simulation: a static analysis of Simulink models. In *Embedded Software and Systems, 2009. ICESS'09. International Conference on*, pages 83–92. IEEE, 2009.

[CMRI12] Sylvain Conchon, Guillaume Melquiond, Cody Roux, and Mohamed Iguernelala. Built-in treatment of an axiomatic floating-point theory for SMT solvers. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories*, pages 12–21, Manchester, UK, 2012.

[Coe95] Tim Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal*, 20(4):129–135, 148, 1995.

[Con12] Sylvain Conchon. *SMT Techniques and their Applications: from Alt-Ergo to Cubicle*. Habilitation thesis, Université Paris-Sud, December 2012.

[Coq14] The Coq Development Team. *The Coq Proof Assistant: Reference Manual, Version 8.4pl4*, 2014.

[Dav72] Philip J. Davis. Fidelity in mathematical discourse: Is one and one really two? *The American Mathematical Monthly*, 79:252–263, 1972.

[DDDM01] David Defour, Florent De Dinechin, and Jean-Michel Muller. Correctly rounded exponential function in double-precision arithmetic. In *International Symposium on Optical Science and Technology*, pages 156–167. International Society for Optics and Photonics, 2001.

[dDLM11] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *Transactions on Computers*, 60(2):242–253, 2011.

[Dek71] Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *LNCS*, pages 53–69. Springer Berlin Heidelberg, 2009.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DK11] Eva Darulova and Viktor Kuncak. Trustworthy numerical computation in Scala. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 325–344, Portland, OR, USA, 2011.

[DK14] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 235–248. ACM, 2014.

[DLP77] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 206–214. ACM Press, 1977.

[DM10] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1):1–20, 2010.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DMC05] G. Dowek, C. Muñoz, and V. Carreño. Provably safe coordinated strategy for distributed conflict resolution. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047*, San Francisco, California, 2005.

[DRT01] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[Edm13] M. G. Edmunds. The Antikythera mechanism and the early history of mechanical computing. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *IEEE Symposium on Computer Arithmetic*, page 79. IEEE Computer Society, 2013.

[EF11] Michael Edmunds and Tony Freeth. Using computation to decode the first known computer. *Computer*, 44(7):32–39, 2011.

[FGP14] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *26th International Conference on Computer Aided Verification*, LNCS, Vienna, Austria, July 2014. Springer.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), June 2007.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[Fil11] Jean-Christophe Filliâtre. *Deductive Program Verification*. Habilitation thesis, Université Paris-Sud, December 2011.

[Fil13] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *LNAI*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 15–29, Seattle, WA, USA, November 2004. Springer.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer.

[Fre08] Goran Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer*, 10(3):263–279, 2008.

[GM74] W. Morven Gentleman and Scott B. Marovich. More on algorithms that reveal properties of floating point arithmetic units. *Communications of the ACM*, 17(5):276–277, 1974.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gol91] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[Goo14] Women who choose computer science – what really matters. Technical report, Google, May 2014. `https://docs.google.com/file/d/0B-E2rcvhnlQ_a1Q4VUxWQ2dtTHM/view?sle=true`.

[Gor00] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, pages 169–186. MIT press, 2000.

[GP06] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *Static Analysis*, pages 18–34. Springer Berlin Heidelberg, 2006.

[GP13] Eric Goubault and Sylvie Putot. Robustness analysis of finite precision implementations. In Chung-Chieh Shan, editor, *Programming Languages and Systems*, volume 8301 of *LNCS*, pages 50–57. Springer International Publishing, 2013.

[GPV12] Eric Goubault, Sylvie Putot, and Franck Védrine. Modular static analysis with zonotopes. In *Static Analysis*, pages 24–40. Springer, 2012.

[Har97] John Harrison. Verifying the accuracy of polynomial approximations in HOL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 137–152, Murray Hill, New Jersey, 1997.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics*, pages 113–130, Nice, France, 1999.

[Har00a] John Harrison. Floating point verification in HOL light: The exponential function. *Formal Methods in System Design*, 16(3):271–305, June 2000.

[Har00b] John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 217–233, Austin, Texas, 2000.

[Har09] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 60–66, München, Germany, 2009.

[Har13] John Harrison. The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50:173–190, 2013.

[Hau96] John R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):139–174, 1996.

[HGBK12] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, pages 131–140, 2012.

[HHK+00] Michael Hemmer, Susan Hert, Lutz Kettner, Sylvain Pion, and Stefan Schirra. Number types. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 edition, 2000.

[Hig02] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. Second edition.

[HIH13] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 279–294, Rennes, France, 2013.

[HKST99] John Harrison, Ted Kubaska, Shane Story, and Ping Tak Peter Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999 Q4, 1999.

[HLB01] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ARITH '01, pages 155–162, Washington, DC, USA, 2001. IEEE Computer Society.

[HLL+12] John Hatcliff, Gary T Leavens, K Rustan M Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys (CSUR)*, 44(3):16, 2012.

[HLQ12] Luke Herbert, K Rustan M Leino, and Jose Quaresma. Using dafny, an automatic program verifier. In *Tools for Practical Software Verification*, pages 156–181. Springer, 2012.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HOL12] HOL4 development team. The HOL System LOGIC. Technical report, University of Cambridge, NICTA and University of Utah, 2012.

[IEE85] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[IEE08] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[IH12] Fabian Immler and Johannes Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In *Interactive Theorem Proving*, pages 377–392. Springer, 2012.

[IM13] Arnault Ioualalen and Matthieu Martel. Synthesizing accurate floating-point formulas. In *24th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 113–116, June 2013.

[Imm14] Fabian Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *NASA Formal Methods*, pages 113–127. Springer, 2014.

[Jac01] Christian Jacobi. Formal verification of a theory of IEEE rounding. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 239–254, Edinburgh, Scotland, 2001. Supplemental proceedings.

[Jac02] Christian Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, Computer Science Department, Saarland University, Saarbrucken, Germany, 2002.

[JB05] Christian Jacobi and Christoph Berg. Formal verification of the VAMP floating point unit. *Formal Methods in System Design*, 26(3):227–266, 2005.

[JLM13] Claude-Pierre Jeannerod, Nicolas Louvet, and Jean-Michel Muller. Further analysis of Kahan's algorithm for 2x2 determinants. *Mathematics of Computation*, 82(284):2245–2264, October 2013.

[Kah86] William Kahan. Miscalculating Area and Angles of a Needle-like Triangle. Unpublished manuscript. `http://www.cs.berkeley.edu/~wkahan/Triangle.pdf`, 1986.

[Kah04] W. Kahan. On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic. Unpublished manuscript. `http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf`, November 2004.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[KK03] Roope Kaivola and Katherine Kohatsu. Proof engineering in the large: formal verification of Pentium®4 floating-point divider. *International Journal on Software Tools for Technology Transfer*, 4(3):323–334, 2003.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Knu97] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

[Kor05] Peter Kornerup. Digit selection for SRT division and square root. *IEEE Transactions on Computers*, 54(3):294–303, 2005.

[LBD03] Ren-Cang Li, Sylvie Boldo, and Marc Daumas. Theorems on efficient argument reductions. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 129–136, Santiago de Compostela, Spain, 2003.

[Lea06] Gary T Leavens. Not a number of floating point problems. *Journal of Object Technology*, 5(2):75–83, 2006.

[Lec35] Maurice Lecat. *Erreurs de mathématiciens des origines à nos jours*. Castaigne, 1935.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LM01] Vincent Lefèvre and Jean-Michel Muller. Worst cases for correct rounding of the elementary functions in double precision. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th Symposium on Computer Arithmetic*, pages 111–118, Vail, Colorado, 2001.

[LM12] Catherine Lelay and Guillaume Melquiond. Différentiabilité et intégrabilité en Coq. Application à la formule de d'Alembert. In *Vingt-troisièmes Journées Francophones des Langages Applicatifs*, Carnac, France, February 2012.

[Mac04] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust.* MIT Press, 2004.

[Mal72] Michael A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Commun. ACM*, 15(11):949–951, November 1972.

[Mar07] Claude Marché. Jessie: an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM.

[May01] Micaela Mayero. *Formalisation et automatisation de preuves en analyses rélle et numérique.* PhD thesis, Université Paris VI, décembre 2001.

[MBdD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser, 2010.

[MDMM13] Érik Martin-Dorel, Guillaume Melquiond, and Jean-Michel Muller. Some issues related to double rounding. *BIT Numerical Mathematics*, 53(4):897–924, 2013.

[Mel12] Guillaume Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012.

[Min95] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report 110167, NASA Langley Research Center, 1995.

[ML96] Paul S. Miner and James F. Leathrum. Verification of IEEE compliant subtractive division algorithms. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 64–78, 1996.

[MLK98] J. Strother Moore, Thomas Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5K86 floating point division program. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[MMNS11] R.M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119 – 161, 2011.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3):12, May 2008.

[MP07] Guillaume Melquiond and Sylvain Pion. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*, 41(1):57–70, 2007.

[MPF14] The MPFR team. *The MPFR library: algorithms and proofs*, 2014. `http://www.mpfr.org/algorithms.pdf`.

[MS13] Evgeny Makarov and Bas Spitters. The Picard algorithm for ordinary differential equations in Coq. In *Interactive Theorem Proving*, pages 463–468. Springer, 2013.

[MSCD05] César Muñoz, Radu Siminiceanu, Víctor Carreño, and Gilles Dowek. KB3D reference manual - version 1.a. Technical Memorandum NASA/TM-2005-213679, NASA Langley, NASA LaRC,Hampton VA 23681-2199, USA, June 2005.

[NJN07] Markus Neher, Kenneth R. Jackson, and Nedialko S. Nedialkov. On Taylor model based integration of ODEs. *SIAM Journal on Numerical Analysis*, 45(1):236–262, 2007.

[NK09] Adam Naumowicz and Artur Korniłowicz. A brief overview of Mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 67–72, München, Germany, 2009.

[NM11] Thi Minh Tuyen Nguyen and Claude Marché. Hardware-dependent proofs of numerical programs. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, LNCS, pages 314–329. Springer, December 2011.

[ORO05] Takeshi Ogita, Siegfried M Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[OZGS99] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating point hardware. *Intel Technology Journal*, 3(1), 1999.

[PA13] Gabriele Paganelli and Wolfgang Ahrendt. Verifying (in-) stability in floating-point programs by increasing precision, using SMT solving. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. SYNASC, 2013.

[PMP+14] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with verifast: Industrial case studies. *Science of Computer Programming*, 82(1):77–97, March 2014.

[PMR14] Olivier Ponsini, Claude Michel, and Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, pages 1–27, 2014.

[Pol98] Robert Pollack. How to believe a machine-checked proof. *Twenty Five Years of Constructive Type Theory*, 36:205, 1998.

[PQ08] André Platzer and Jan-David Quesel. Keymaera: A hybrid theorem prover for hybrid systems. In *Automated Reasoning*, pages 171–178. Springer, 2008.

[Pri91] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 132–143. IEEE, 1991.

[ROS98] John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[Rou14] Pierre Roux. Innocuous double rounding of basic arithmetic operations. *Journal of Formalized Reasoning*, 7(1), 2014.

[RS06] Erik Reeber and Jun Sawada. Combining ACL2 and an automated verification tool to verify a multiplier. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 63–70. ACM, 2006.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[Rus99] David M. Russinoff. A mechanically checked proof of correctness of the AMD-K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75, 1999.

[Rus00] David M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. *LNCS*, 1954:3–36, 2000.

[Rus13] David M. Russinoff. Computation and formal verification of SRT quotient and square root digit selection tables. *IEEE Transactions on Computers*, 62(5):900–913, 2013.

[RW10] Philipp Rümmer and Thomas Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[RZBM09] Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT*, 49(2):419–431, June 2009.

[SG02] Jun Sawada and Ruben Gamboa. Mechanical verification of a square root algorithm using Taylor's theorem. In Mark D. Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 274–291. Springer Berlin Heidelberg, 2002.

[She97] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.

[SST05] Eric M. Schwarz, Martin Schmookler, and Son Dao Trong. FPU implementations with denormalized numbers. *IEEE Transactions on Computers*, 54(7):825–836, 2005.

[Ste74] Pat H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.

[The14] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 edition, 2014.

[Try93] A. Trybulec. Some features of the Mizar language. In *Proceedings of the ESPRIT Workshop*, Torino, Italy, 1993.

[Tuc99] Warwick Tucker. The Lorenz attractor exists. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 328(12):1197–1202, 1999.

[Tuc11] Warwick Tucker. *Validated numerics: a short introduction to rigorous computations*. Princeton University Press, 2011.

[Vel69] Gerhard W. Veltkamp. ALGOL procedures voor het rekenen in dubbele lengte. RC-Informatie 21, Technische Hogeschool Eindhoven, 1969.

[Yu92] Yuan Yu. *Automated proofs of object code for a widely used microprocessor*. PhD thesis, University of Texas at Austin, 1992.

[ZKR09] Karen Zee, Viktor Kuncak, and Martin Rinard. An integrated proof language for imperative programs. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2009.

[ZSR14] Timothy K. Zirkel, Stephen F. Siegel, and Louis F. Rossi. Using symbolic execution to verify the order of accuracy of numerical approximations. Technical Report UD-CIS-2014/002, Department of Computer and Information Sciences, University of Delaware, 2014.

[Zum06] Roland Zumkeller. Formal global optimisation with Taylor models. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130 of *LNCS*, pages 408–422. Springer Berlin Heidelberg, 2006.