Formal Verification for Numerical Computations, and the Other Way Around

Guillaume Melquiond

Habilitation à Diriger des Recherches

présentée à l'Université Paris-Sud

Spécialité : Informatique

Guillaume Melquiond

Formal Verification for Numerical Computations, and the Other Way Around

Soutenue le 1er avril 2019 devant la commission d'examen constituée de

Luc Jaulin	rapporteurs
David Monniaux	
Warwick Tucker	
Yves Bertot	examinateurs
Florent Hivert	
Claude Marché	
Jean-Michel Muller	
Sylvie Putot	

Préface

J'ai soutenu ma thèse le 21 novembre 2006, thèse qui s'intitulait « De l'arithmétique d'intervalles à la certification de programmes ». Si je devais la réécrire aujourd'hui, j'emploierais plutôt le mot vérification que certification, mais cela ne changerait pas grand chose à la pertinence et à l'actualité de son sujet. Néanmoins, douze années ont passé et mes thématiques de recherche se sont diversifiées. Il est donc temps pour moi de m'attaquer à l'habilitation. Et comme la science n'avance pas seulement à coup de publications révolutionnaires dans des congrès, ce document d'habilitation est aussi l'occasion pour moi de mettre en perspective mes travaux de recherche et de revenir sur les bonnes idées qui ont su résister au nombre des années mais aussi sur celles qui se sont plus tard avérées être des impasses.

Remerciements

Avant d'entrer dans le vif du sujet, je tiens à remercier Luc Jaulin, David Monniaux et Warwick Tucker pour avoir accepté d'être rapporteurs de cette habilitation.

J'aimerais remercier Sylvie Boldo, mon *alter ego* en matière de recherche en arithmétique des ordinateurs et en preuves formelles. J'aimerais aussi remercier Christine Paulin and Claude Marché de m'avoir accueilli dans leur équipe de recherche et cela malgré le large fossé qui semblait exister entre nos thématiques de recherche.

Il y a aussi toutes les personnes avec qui j'ai eu de longues et fructueuses discussions et sans qui nombre des travaux présentés dans ce document n'existeraient pas. Je tiens donc à remercier Andrei Paskevich, Arnaud Tisserand, Assia Mahboubi, Bruno Salvy, Catherine Lelay, César Muñoz, Christoph Lauter, Claude-Pierre Jeannerod, Clément Fumex, Cody Roux, Daisuke Ishii, Enrico Tassi, Érik Martin-Dorel, Florent de Dinechin, François Bobot, François Clément, Hervé Brönnimann, Jacques-Henri Jourdan, Jean-Christophe Filliâtre, Jean-Michel Muller, Laurence Rideau, Laurent Théry, Marc Daumas, Marc Mezzarobba, Maxime Dénès, Micaela Mayero, Mohamed Iguernelala, Nathalie Revol, Nicolas Brisebarre, Nicolas Louvet, Pascal Cuoq, Paul Zimmermann, Pierre Roux, Raphaël Rieu-Helft, Sylvain Conchon, Sylvain Pion, Thomas Sibut-Pinote, Xavier Leroy et Yves Bertot. Et si j'ai oublié quelqu'un dans cette longue liste, ce qui est vraisemblablement le cas, j'en suis vraiment navré. J'aimerais aussi remercier celles qui m'ont apporté leur soutien administratif au cours de cette période : Katia Évrat et Régine Bricquet.

Comme ils n'ont pas encore eu l'occasion d'apparaître, je tiens à remercier Florent Hivert et Sylvie Putot pour avoir accepté de participer à la commission d'examen de cette habilitation.

Et finalement, je tiens à remercier les membres de mon équipe et mon entourage proche sans qui je n'aurais pas pu effectuer toutes ces années de recherche dans de bonnes conditions.

Résumé

Mes travaux de recherche visent à favoriser l'utilisation des outils de vérification formelle et en particulier les approches déductives dans le domaine du calcul numérique. En effet, les programmes utilisent rarement l'arithmétique réelle pour effectuer leurs calculs et ils se rabattent généralement sur des arithmétiques approchées telles que l'arithmétique à virgule flottante, pour des raisons d'efficacité. L'emploi de ces arithmétiques entraîne des erreurs de calcul qui peuvent empêcher un programme de s'exécuter comme prévu. Il est donc important de les prendre en compte pour garantir la correction des programmes qui effectuent des calculs numériques. La subtilité de l'arithmétique à virgule flottante rend malheureusement l'exercice périlleux, d'où la nécessité de développer des méthodes qui permettent de rendre sûr ce travail de vérification.

Arithmétique d'intervalles

L'arithmétique d'intervalles consiste à représenter une valeur réelle idéale en l'encadrant par deux valeurs approchées. Il s'agit d'une façon simple et plutôt efficace de garantir des propriétés sur des nombres réels sans pour autant s'appuyer sur une arithmétique en précision arbitraire. Pour populariser l'arithmétique d'intervalles, j'ai écrit la bibliothèque C++ générique Boost.Interval [BMP06b] qui est à la base de l'outil Gappa [DM10] mais qui a aussi servi à invalider une conjecture mathématique par le calcul [MNZ13]. J'ai aussi contribué à la normalisation de l'arithmétique d'intervalles [EM09, IEE15] et à des propositions en ce sens pour la norme C++ [BMP06c, MP09].

Même si, sur le principe, l'arithmétique d'intervalles garantit les propriétés des valeurs qu'elle encadre, il subsiste le risque que son implémentation ou celle de l'arithmétique sous-jacente sur les bornes approchées souffrent de bogues. Pour augmenter la confiance dans les preuves obtenues par arithmétique d'intervalles, j'ai développé des bibliothèques de calcul en virgule flottante et par intervalles qui s'exécutent raisonnablement efficacement dans la logique du système formel Coq et j'ai formellement prouvé leur correction [Mel08b, Mel12]. Cela a donné la bibliothèque CoqInterval qui contient une tactique capable de prouver automatiquement et formellement des propriétés sur des fonctions à valeurs réelles, par exemple des bornes sur l'erreur commise quand un polynôme d'approximation est utilisée à la place d'une fonction trigonométrique.

L'arithmétique d'intervalles présente un inconvénient majeur : les encadrements qu'elle calcule sont parfois si larges qu'ils n'apportent aucune information utile sur les valeurs encadrées. L'un des défis est donc d'affiner les encadrements sans pour autant faire exploser les temps de calcul. Cela a conduit à enrichir CoqInterval d'une arithmétique à base de modèles de Taylor [MDM16]. Cette amélioration a permis de grandement automatiser la vérification

formelle de l'implémentation en virgule flottante de fonctions élémentaires [BM17]. Cette amélioration a aussi rendu possible le calcul efficace et vérifié d'encadrements, non plus seulement de fonctions, mais aussi d'intégrales propres [MMSP16].

L'étape suivante consiste à s'attaquer à des intégrales impropres. De nombreuses preuves mathématiques modernes (en combinatoire ou en théorie des nombres par exemple) s'appuient sur le calcul numérique approché d'intégrales impropres (pour cause d'absence de formules closes) et souffrent donc du risque d'être invalidées si les calculs s'avèrent incorrects. L'objectif est donc d'étendre le formalisme à base d'arithmétique d'intervalles au calcul de telles intégrales impropres, la difficulté étant que les méthodes numériques usuelles ne s'appliquent plus et qu'une dose de calcul symbolique est alors nécessaire [MMSP19]. Un autre axe de recherche concerne le calcul d'encadrements de fonctions qui ne sont plus définies explicitement par une formule close mais implicitement par une équation différentielle. À long terme, l'objectif est de convaincre les mathématiciens qu'ils n'ont plus besoin de s'appuyer sur des outils non vérifiés parce que les méthodes vérifiées sont souvent aussi puissantes.

Arithmétique à virgule flottante

L'arithmétique d'intervalles garantit ses résultats en calculant des valeurs approchées qui encadrent les valeurs exactes. L'usage veut malheureusement, pour des raisons d'efficacité, que l'on ne calcule qu'une seule valeur flottante en espérant que la valeur exacte n'en est pas trop éloignée. Une analyse de l'erreur commise permet néanmoins de retrouver l'information nécessaire pour garantir la correction du résultat. Les multiples spécificités de l'arithmétique à virgule flottante peuvent rendre une telle analyse non seulement fastidieuse mais aussi incorrecte.

Pour rendre les analyses de code flottant correctes, je m'intéresse à la formalisation de cette arithmétique. En plus de la partie calculatoire présente dans CoqInterval, j'ai développé une formalisation générique de l'arithmétique à virgule flottante nommée Flocq [BM11]. Armé de cette bibliothèque, il devient alors possible de s'attaquer à la vérification formelle en Coq de nombreux algorithmes subtils [BM17] : discriminant précis [BDKM06], prédicats géométriques robustes [MP07], émulation du *fused-multiply-add* [BM08] et autres [RZBM09, MDMM13]. Ces travaux sur l'arithmétique à virgule flottante ont aussi été l'occasion d'écrire plusieurs livres [MBdD$^+$10, BM17, MBdD$^+$18] et de participer à la normalisation de cette arithmétique [IEE08].

Cependant, il ne suffit pas de pouvoir vérifier formellement les preuves d'algorithmes flottants, encore faut-il qu'elles soient humainement réalisables. Pour cela, je me consacre à développer des méthodes pour en automatiser la majeure partie. Le cœur de cet effort est l'outil Gappa [DM10] : à l'aide d'arithmétique d'intervalles et d'une base de théorèmes sur l'arithmétique flottante, il cherche à reproduire les preuves d'analyse d'erreur telles qu'elles auraient pu être écrites par un humain, simplifiant ainsi considérablement le travail de ce dernier [dDLM11]. Qui plus est, ces preuves sont vérifiables par Coq ; l'outil peut même être invoqué depuis l'assistant de preuves lui-même [BFM09].

Gappa est conçu pour trouver des preuves de propriétés arithmétiques très pointues sur des codes complexes mais courts. Il se prête assez mal à la vérification de propriétés basiques sur des codes simples, longs et ne parlant pas que d'arithmétique flottante. C'est pourquoi je m'intéresse aussi aux prouveurs de type SMT (satisfiabilité modulo théories) et comment les adapter aux propriétés flottantes [CMRI12, CIJ$^+$17]. Cela passe entre autres par leur

amélioration concernant la preuve de propriétés liées aux entiers et aux rationnels [BCC+12]. L'un de mes axes de recherche est de développer plus avant cette notion de preuve automatique mixant flottants et réels dans le cadre SMT. La difficulté majeure réside dans l'opposition des philosophies qui sous-tendent les outils : raisonnement booléen pour les prouveurs SMT contre preuve guidée par le but pour Gappa.

Analyse réelle

Que ce soit pour l'arithmétique d'intervalles ou l'arithmétique flottante, il existe une arithmétique sous-jacente qui est l'arithmétique réelle. Cette arithmétique a été formalisée au sein de nombreux assistants de preuve [BLM16] mais on en atteint tôt ou tard les limites quand on cherche à vérifier des algorithmes de calcul numérique. Par exemple, dans le cas de Coq, la formalisation de base ne permet pas vraiment de manipuler des séries entières, des intégrales, des équations différentielles, etc. C'est dans l'optique de rendre plus facile la preuve formelle de théorèmes d'analyse que j'ai co-encadré la thèse de Catherine Lelay avec Sylvie Boldo. Cela a conduit au développement de la bibliothèque Coquelicot, une extension conservative de la bibliothèque standard de Coq [BLM15].

Ce type de formalisation a permis de vérifier un programme de résolution d'une équation aux dérivées partielles : les valeurs calculées par le programme sont effectivement proches des valeurs qu'aurait prises le système réel et ce malgré les erreurs flottantes et les erreurs de discrétisation [BCF+14]. Sur un sujet connexe, je me suis aussi intéressé à la vérification de propriétés de sûreté pour des automates hybrides, c'est-à-dire des systèmes alternant transitions discrètes et comportement continu [IMN13].

Pour pouvoir vérifier des programmes et algorithmes plus compliqués, il sera nécessaire de généraliser les formalismes concernant les équations différentielles, les séries entières et les comportements asymptotiques. À long terme, l'objectif est d'offrir des bibliothèques formelles riches et disposant de suffisamment d'outils automatiques pour que le passage d'une preuve papier à une preuve formelle ne constitue plus un obstacle insurmontable.

Outils vérifiés

Cette volonté de vérifier des programmes qui manipulent plus que de l'arithmétique flottante m'a conduit à participer au développement de la plateforme de vérification de programmes Why3 [BFM+13]. Je me suis aussi intéressé à la vérification formelle des compilateurs, en particulier en matière d'arithmétique flottante [BJLM15]. En effet, à quoi bon vérifier qu'un programme est correct s'il se retrouve compilé incorrectement ?

Le rôle de la plateforme Why3 consiste principalement à transformer la propriété de correction d'un programme en un ensemble d'énoncés mathématiques dont la preuve est déléguée à des outils externes telles que Gappa, Coq ou des prouveurs SMT. L'un de mes axes de recherche est de simplifier le travail de l'utilisateur et de réduire la dépendance de Why3 sur des outils externes, sans pour autant réduire la confiance en Why3. C'est dans ce cadre que se place la thèse de Raphaël Rieu-Helft, que j'encadre. Une des approches explorées consiste à transposer à Why3 le paradigme de la preuve par réflexion [MRH18]. Je souhaite appliquer cette méthode à la vérification formelle de bibliothèques de calcul sur grands entiers, un composant critique des logiciels de calcul formel ou de cryptographie. La particularité d'une bibliothèque comme GNU Multi-Precision (GMP) est que sa quête acharnée de performance

a entraîné l'utilisation d'algorithmes extrêmement subtils, ce qui constitue donc un défi pour les outils de vérification. Qui plus est, il ne s'agit pas juste de vérifier les algorithmes, il faut aussi pouvoir obtenir une bibliothèque C compétitive avec GMP [RHMM17]. En particulier, il est important de pouvoir spécifier finement le comportement des programmes avec pointeurs, sans pour autant que cela rende la vérification pénible.

Un autre de mes axes de recherche concerne l'assistant de preuves Coq. La bibliothèque CoqInterval le pousse déjà dans ses retranchements en matière de calculs numériques. Si l'on souhaite automatiser la preuve de théorèmes encore plus compliqués, il faudra trouver un moyen d'augmenter ses capacités de calcul et là encore il faudra le faire sans pour autant réduire la confiance en l'outil.

Contents

Chapter 1

Introduction

1.1 The trouble with computer arithmetic

Super Mario 64 is a video game released in 1996 for the *Nintendo 64* console. As most platform games, the objective for the player is to lead a character to the exit of a level, by making it jump from platform to platform, collecting objects along the way. Soon enough, speedrunners, a niche category of players, added various handicaps to make the game more challenging. One such handicap was to finish the game without ever pressing the A button. Since pressing A makes the character jump, this might seem like an insurmountable challenge for a platform game. Yet, imaginative speedrunners found numerous ways to exploit bugs in the game in order to build vertical momentum without pressing this button. Eventually, they were able to finish the game without ever jumping, except for one specific level. Despite their best efforts, they could not find any bug to skip the last remaining jump. But everything changed in Spring 2018, when a player forgot to turn off the game for a few hours and noticed that a mobile platform in that level had imperceptibly drifted upward. Speedrunners took advantage of this: letting the game run for several days was enough for the platform to lift Mario high enough. That was it, they had fully broken this archetypal game.

What makes this anecdote interesting is the reason for the platform drifting. First, players quickly noticed that, in the Nintendo 64 version of the game, the platform is not drifting. This happens only in the version released in 2006 for the *Wii Virtual Console*. This is an emulator designed so that owners of the Wii console could play (and, more importantly, buy) older games. While the code of the 1996 and 2006 releases of the game is basically the same, it behaves differently. The difference lies in the way the original hardware and the emulated hardware handle *floating-point arithmetic*. Let us explain. To update the height of the platform at frame n, the game executes $y \leftarrow y + R\sin(\omega n)$. For the sake of performance and memory footprint, these computations are not computed with infinite accuracy. In fact, the game stores the content of the y variable using the *binary32* floating-point format, which cannot represent the result of $y + R\sin(\omega n)$ exactly. So, the update code should rather be understood as $y \leftarrow \square(y + R\sin(\omega n))$, where $\square$ is some kind of virtual operator that chooses a *binary32* number close enough to the mathematical result. This operator is a *rounding* operator.

The main issue lies in the definition of "close enough". On the original hardware, arithmetic operations computed the floating-point number the nearest to the infinitely precise result. So, some rounding errors occurred at each frame, but luckily, these errors averaged

out in the long run and they caused no platform drift. On the emulated hardware, however, rounding was to zero, *i.e.*, of the two floating-point numbers enclosing the infinitely precise result, the one the closest to zero was chosen. As a consequence, since y was of constant sign, the rounding error caused at each frame was always directed in the same direction, causing the platform to slowly drift toward height 0 over time [ABC18].

This example of rounding errors causing an unexpected behavior in a program might seem especially innocuous. Unfortunately, that is not always the case. In 1991, a strikingly similar bug led to the death of 28 persons. During the first Gulf War, the U.S. Army had installed Patriot defenses to intercept incoming Scud missiles. The tracking software was counting time as an integer n in unit of tenths of a second. Then, when performing computations, it would first turn it into a floating-point number using $t \leftarrow n \cdot \square(0.1)$. This introduces a rounding error: the larger n is, the further t is from the actual time. *A priori*, this error should not have caused any issue, since it is not the value of t which matters. What matters is the difference $t - t'$ with some previously computed value $t' \leftarrow n' \cdot \square(0.1)$. Since n and n' are close in practice, the rounding error is negligible. Unfortunately, to cope with the high velocity of Scud missiles, the tracking software had been modified to make some (but not all) computations more accurate, which means that the rounding operator was no longer the same everywhere. As a consequence, even the difference $t - t'$ drifted over time. Indeed, we have $t - t' = (n - n') \cdot \square(0.1) + n' \cdot (\square(0.1) - \square'(0.1))$. Unless $\square(0.1)$ is equal to $\square'(0.1)$, even if $n - n'$ stays bounded, $t - t'$ grows over time. After letting the system run for a few days, this caused an anti-missile to miss its target by a fraction of a second [Ske92].

Let me stress one important point: the bug occurred because some parts of the system were made more accurate. Another instance of this phenomenon can be found in the folklore, as it is well-known that, if a catastrophic cancellation occurs during a floating-point subtraction, then the round-off error explodes, yet the result is computed exactly. These examples show that numerical computations are not as intuitive as developers might expect. In particular, the correctness of an algorithm cannot generally be estimated at first glance, nor at any later glance.

The case of the Intel Pentium `FDIV` was a wake-up call. This floating-point divisor design was based on a radix-4 SRT algorithm, which uses a bidimensional lookup table to compute a quotient digit among -2, -1, 0, 1, 2. Unfortunately, five entries of the table (among about 1400 entries) were incorrect, causing 0 to be used instead of 2 [CT95]. These entries were difficult to reach by random testing, though, as the ratio of failure was one over 10 billion pairs of floating-point inputs. This led Pratt to write the following paragraph on the difficulty of testing in that case [Pra95].

> One thinks of testing as being as good as verification if one could test all possible cases. As a weakened version of this, a comprehensive test should exercise every device and/or line of code in the system. The Pentium bug reveals a serious limitation of this approach. There is of course no data that can exercise unreachable code or table entries. Thus if one believes that the five "missing" entries are unreachable, then no attempt will be made to produce a test for this case.

The large input space combined with the lack of intuitiveness of floating-point algorithms push for their detailed mathematical verification. Among the early attempts of this approach, one can cite Holm's work which combined a floating-point formalism with Hoare's logic in order to check numerical algorithms [Hol80]. Barrett later used the Z notation to specify

the IEEE-754 standard and refine it to a hardware implementation [Bar89]. One can also cite Priest's work as an example of a generic formalism for designing guaranteed floating-point algorithms [Pri91]. But all of these early works were mostly based on detailed pen-and-paper mathematical proofs and the assistance of computers was limited. Unfortunately, computer arithmetic in general, and floating-point arithmetic in particular, might make things so contrived that corner cases can easily be missed. This led Pratt to add:

> Nevertheless one may feel reassured after having proved manually that an algorithm works in every detail, and hence attach little incremental value to a machine-checked proof. This opens the way to $475,000,000 errors, Intel's estimate of the cost of the bug.

While we now understand why the `FDIV` operator was producing incorrect results, we do not know how the table entries came to be wrong. So, it is hard to be as confident as Pratt that a machine-checked proof would have avoided that bug. Still, this led the industry to formally verify some floating-point hardware designs. The various works around the floating-point units embedded in AMD-K5 processors are an illustration of this effort. Moore, Lynch, and Kaufmann were interested in the correctness of the division algorithm [MLK98], while Russinoff tackled the arithmetic operators at the RTL level [Rus98, Rus00] and the square root at the microcode level [Rus99]. All these proofs were based on a formalism written for the ACL2 first-order proof assistant.[1]

These works are only a fraction of what has been done around the verification of hardware designs using formal systems. Around the same time, people also started to apply formal systems to the verification of floating-point software. One can cite some early works of Harrison in HOL Light[2] [Har99, Har06]. He also verified the correctness of the division operator found on Intel Itanium processors [Har00b]. Since fused multiply-add (FMA) is the only floating-point operator provided by these processors, division has to be implemented in software by a sequence of FMA. As such, the approach is similar to AMD-K5's microcoded division, and so is the verification. Harrison also proved the correctness of a floating-point implementation of the exponential function [Har97a] and of some trigonometric functions [Har00a].

1.2 Theorems and computational proofs

While formal systems considerably increase the trust one can have in a proof, be it that of an algorithm, of a hardware design, or of a mathematical theorem, they do not come without downsides. Of special importance is the fact that they also greatly increase the proof effort. Indeed, these systems mostly work at a syntactic level: if both properties A and "if A then B" hold, then property B holds too, whatever the meaning of A and B. This makes it possible for a computer to mechanically check that a proof is correct, but at the expense of a great deal of verbosity for the proof writer. Indeed, the writer can no longer assumes that the reader is a human clever enough to fill the blanks in the proof. To alleviate this issue, it is important that large parts of the proof can be automated, *i.e.*, the computer should search for the proof itself rather than having the user write it *in extenso*.

The use of a computer to perform parts of a mathematical proof is nothing specific to formal verification. In the recent years, several mathematical theorems were proved, whose proof was too long to be entirely written by hand. A first illustration is the theorem stating

[1] `http://www.cs.utexas.edu/users/moore/acl2/`

[2] `http://www.cl.cam.ac.uk/~jrh13/hol-light/`

that only 15 kinds of convex pentagons can pave the plane. A hand-written proof first partitions convex pentagons into 371 families that are candidates for paving the plane. Then, a program tries each of these families by placing pentagons as long as it can, and by backtracking otherwise. As the program always terminates, except for the 15 families already known, this concludes the proof [Rao17].

Another example is the fact that the smallest surface enclosing two equal volumes is composed of two truncated spheres joined by a disk, *i.e.*, a *double bubble*. Again, the proof starts by explaining why some reduced search space can be considered. Then, a program performs a *bisection* on this space, ending up with 15,016 cases, which it can all handle [HS00]. What brings this example closer to the topic of computer arithmetic is the fact that every case is handled by numerically estimating the value of a few integrals by performing floating-point arithmetic. To prevent round-off errors from polluting the estimations, floating-point computations are performed through *interval arithmetic*.

There even exist some theorems that have been verified inside a formal system, even though their proofs rely on the execution of a program. In that case, either the program is emulated inside the formal system, or it runs outside and generates proofs that are verified by the formal system. There might even be some mix of the two approaches: an unverified external program generates a certificate that is checked by a verified program emulated inside the formal system. An example of this is the proof that Schur's fifth number is equal to 160. It means that, whatever the partitioning of $\{1, \ldots, 161\}$ into 5 subsets, one of them contains three integers a, b, and c, such that $a + b = c$. The external program took 14 years of CPU time to generate a certificate, which was then checked by a program verified using ACL2 and running for 36 years of CPU time [Heu18].

That proof is purely combinatorial, but we can also find proofs that are both formally verified and based on intensive numerical computations. An example is the formal proof of Kepler's conjecture, which gives the optimal asymptotic density when packing unit spheres. As with the previous examples, the proof starts by reducing the problem to many subproblems, except that, this time, this reduction proof is formally verified using HOL Light. Part of these subproblems deal with the global optimization of some functions. This is where computer arithmetic matters, as this optimization is performed using floating-point arithmetic, interval arithmetic, and, for the sake of verification speed, *Taylor models*. Note that, for the corresponding arithmetic operators to be used, they had to be implemented and verified in HOL Light first. In the end, the verification of all the inequalities related to global optimization took about one year of CPU time [HAB+15].

What makes this example interesting is that it was not a formal proof originally. It started as a hand-written proof accompanied with some C++ programs to perform the global optimization. The six papers of this work were reviewed by at least 13 people over a period of three years. Despite this huge amount of scrutiny, the editors never reached full confidence in the proof, or, as Lagarias summarized [Lag11]:

> The nature of this proof, consisting in part of a large number of inequalities having little internal structure, and a complicated proof tree, makes it hard for humans to check every step reliably.

This lingering doubt led Hales to turn his original proof into a fully formal one. Lagarias had the following to say about it:

> Constructing a formal proof of the Kepler Conjecture involves not only logic, it is also a giant software project. [...] Formalization of a proof can also uncover gaps in a pre-formal proof. This was the case here. The initial process of formalizing the proof tree of the published proof in the six papers of the Kepler Conjecture did uncover one logical gap in the published proof.

At this point, some pattern is emerging. On one side, we have programs (or hardware) and we might want to guarantee that they compute the expected values. On the other side, we have mathematical theorems and we might want to execute programs to help us prove that these theorems hold. In both cases, we want to use formal proofs to increase the confidence. We also want as much automation as possible, to bring the proof effort to the level of a hand-written proof, if not lower.

My research interests revolve around the formal verification of algorithms, especially those related to computer arithmetic, *e.g.*, floating-point arithmetic, interval arithmetic, and arbitrary-precision integer arithmetic. I am also interested in developing automated methods or frameworks that make the verification of such algorithms much more amenable. Finally, I am interested in increasing the confidence in mathematical results.

1.3 Outline

Chapter 2 presents some concepts that will recurrently appear in this document. The first one is floating-point arithmetic, or rather an abstraction of it that is suitable both for proofs and computations (Section 2.1). Once we have floating-point arithmetic, we can build on top of it interval arithmetic, which is a way to approximate real computations in a safe-by-construction way (Section 2.2). Finally, the last concept to be presented is formal verification, and more precisely some tools, *e.g.*, Coq, that will be extensively used, as well as computational reflection, an efficient way of reasoning (Section 2.3).

Chapter 3 illustrates the use of verified programs to prove mathematical theorems. The goal is to compute some guaranteed numerical bounds on the value of an improper definite integral that appears in Helfgott's proof of the ternary Goldbach conjecture. This chapter follows a bottom-up approach. We start from a Coq formalization of real analysis that is rich enough to define what an integral is, among other things (Section 3.1). Since real numbers are not that suitable for computations, we then define floating-point numbers and their properties. That is where the first complicated programs appear, since we need the floating-point operators to be effective (Section 3.2). While effective, they are a bit cumbersome when it comes to proving properties on real numbers, so we formalize an interval arithmetic on top of them (Section 3.3). In its most naive form, the theorems about real-valued expressions that interval arithmetic can automatically prove are not that interesting, so we incorporate a few improvements (Section 3.4). At that point, we have enough programs and correctness proofs to tackle the computation of Helfgott's integral in a formally verified way (Section 3.5).

While Chapter 3 relies on some floating-point algorithms, the use of directed rounding makes them mostly correct by construction. So, the verification effort actually related to floating-point arithmetic is relatively light. Chapter 4 looks at much more intricate algorithms. This time, the goal is to verify a floating-point function that approximates exponential, *i.e.*, its computed result is provably close to the ideal result. While the chosen function is not as complicated as the one found in CRlibm, it still exhibits some of the challenges that

occur when verifying CRlibm. First, we show how to automatically and formally bound the method error, which ignores any rounding error (Section 4.1). Then, we tackle the round-off error. Again, the main matter is to automate the verification as much as possible, due to the tediousness of formal proofs when doing forward error analysis (Section 4.2). Since the algorithms are expressed in a high-level language and are meant to be executed on floating-point hardware units, formally verifying the algorithms (and the units) is not sufficient, we also have to take the compiler into account. We do so by giving a precise semantics to floating-point operations and by formally proving that the compiler preserves it (Section 4.3). Elementary functions are representative of a large class of small but intricate floating-point code. In some cases, however, we want the result computed by the floating-point algorithm to be exactly equal to the ideal result, or at least we want to detect that this might not be the case, which introduces a few subtleties. We use reliable geometric predicates as an example (Section 4.4). Up to this point, the automated approaches have been dedicated to floating-point algorithms, hence quite *ad hoc*. In particular, anything unrelated to floating-point arithmetic, *e.g.*, arrays, has been ignored. So, it is interesting to see how these aproaches translate to a more generic verification framework such as SMT solvers (Section 4.5).

Chapter 5 presents some of the research works that do not fit in the previous chapters. First, I did not just work on automating the formal verification of bounds on real-valued expressions and round-off errors, I also got interested in linear arithmetic as well as hybrid systems (Section 5.1). Second, it is not always about verifying some existing intricate floating-point algorithms, sometimes it is about inventing brand new ones (Section 5.2). Finally, some larger applications are worth mentioning, from the verification of a numerical scheme for the wave equation to the implementation of a formally verified library for arbitrary-precise integers (Section 5.3).

Almost every section of this document is followed by an *assessment* that evaluates the interest of my work, presents some related works, and details some potential improvements. Chapter 6 gives some more general perspectives on my research.

1.4 Chronology and context

While this document is organized along two main themes, the way my research work evolved during the last fifteen years is not that simple. There was a lot of back and forth and it was hardly ever done in isolation, as I got to supervise the PhD theses of Catherine Lelay (2011–2015, co-supervisor: Sylvie Boldo) and Raphaël Rieu-Helft (2017–). I also collaborated with several postdoctoral students: Daisuke Ishii (2011), Cody Roux (2011–2012), Érik Martin-Dorel (2013–2014), and Pierre Roux (2014).

So, this section tries to put things into context, by detailing when the various topics were researched and which PhD and postdoctoral students I advised on these topics. Figure 1.1 summarily shows how my publications and tools are related. (This figure also shows that, even if graph layout is an NP-complete problem, we should still strive to improve the heuristics used to automatically lay graphs out.)

One of my earliest work was the Boost.Interval library, a generic C++ library for interval arithmetic [BMP03, BMP06b]. This arithmetic offers a simple yet reliable way of computing bounds on real-valued expressions, and thus to (dis)prove mathematical conjectures [MNZ13]. Designing Boost.Interval was an opportunity to consider features that would make reliable

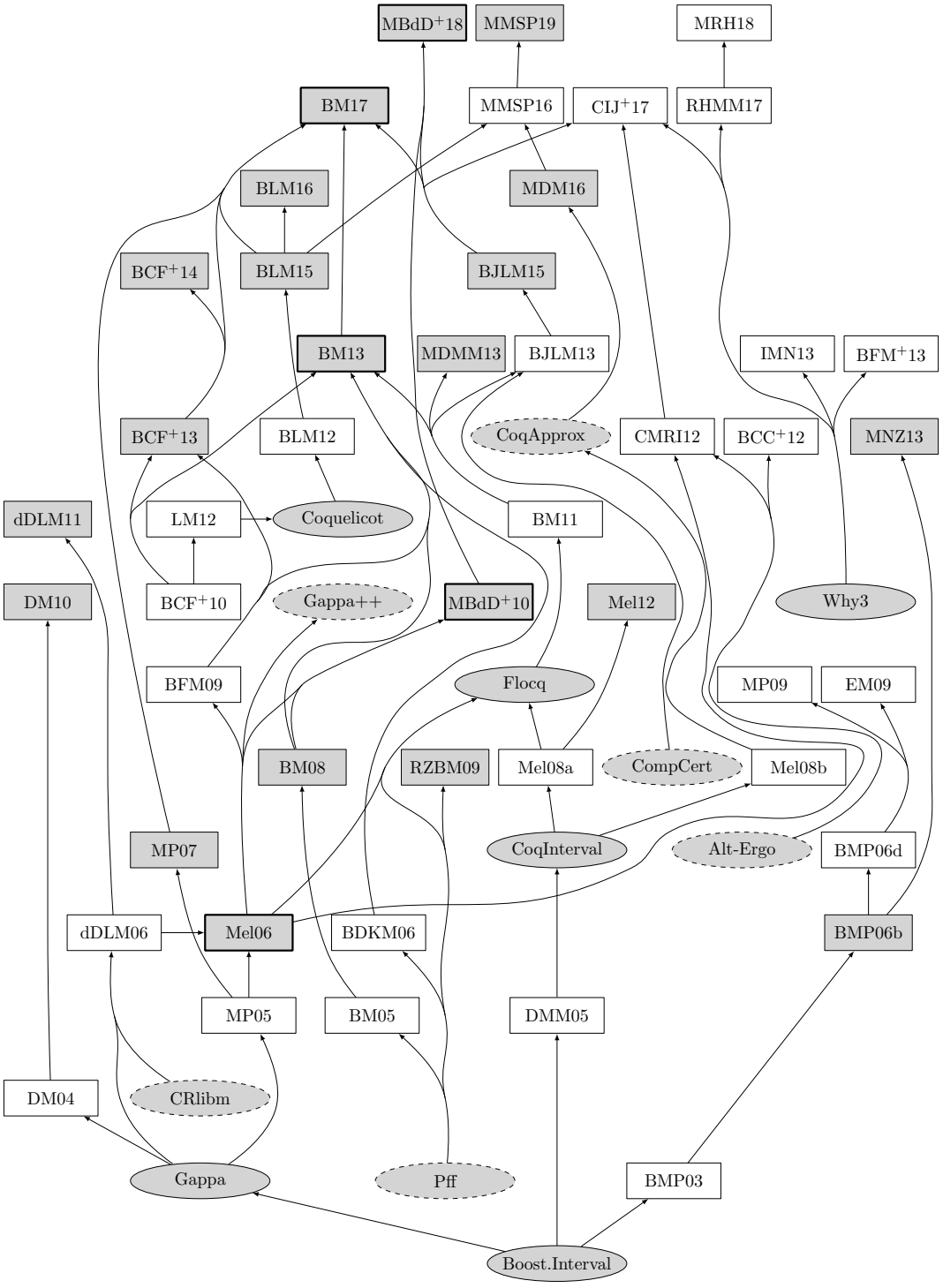


Figure 1.1 – Rectangular nodes are all my publications from 2002 at bottom to 2018 at top. Plain ones represent conference articles; filled ones are for journal articles; and the ones with thick borders are for much larger publications, *e.g.*, books. Ellipsoidal nodes represent related tools and libraries, with a solid border when I am a major contributor.

computations easier to achieve and possibly standardize them [BMP06a, BMP06c, BMP06d, EM09, MP09].

Soon after its creation, I used Boost.Interval for two tools. The first one was the Gappa tool, which uses interval arithmetic to verify properties of floating-point algorithms, especially bounds on round-off errors [DM04, Mel06, DM10]. Gappa has been mainly used to verify the implementation of elementary functions, *e.g.*, in CRlibm [dDLM06, dDLM11], but it has also served in computational geometry [MP05, MP07]. Gappa is dedicated to verifying floating-point properties, but there exist automatic verification tools that are more generic, *e.g.*, SMT solvers. So, we worked on making the Alt-Ergo solver support floating-point arithmetic [CMRI12, CIJ+17], in part during Cody Roux's postdoctoral stay. Alt-Ergo's support for linear integer arithmetic was also improved [BCC+12]. Moreover, the postdoctoral stay of Daisuke Ishii was the opportunity to experiment with automated verification for hybrid systems [IMN13].

The other early use of Boost.Interval was an oracle that generated proofs based on interval arithmetic with rational bounds for the PVS formal system [DMM05]. The oracle mechanism was later dropped and the rational arithmetic was replaced by a floating-point arithmetic, which led to the development of the CoqInterval library for the Coq formal system [Mel08b]. People from the CoqApprox project started to use CoqInterval in order to compute guaranteed polynomial approximations of univariate expressions using Taylor models. These polynomial approximations proved to be a good way to improve the efficiency of interval arithmetic in a formal setting, so that work was later merged into CoqInterval during Érik Martin-Dorel's postdoctoral stay [MDM16].

The Gappa tool does not just verify numerical properties. It is also able to generate formal proofs of them, which Coq can then double-check thanks to a formalization of floating-point arithmetic [BFM09]. This means that, at a point, there were three unrelated Coq formalizations of floating-point arithmetic: the one from CoqInterval [Mel08a, Mel12], the one from Gappa, and the one from the preexisting Pff library. These three formalizations were generalized into a single framework, Flocq [BM11]. Pff and then Flocq have been used to prove numerous properties of floating-point arithmetic [BM05, BDKM06, BM08, RZBM09, MDMM13]. The Flocq library was later extended, so as to help implementing and verifying the floating-point support of the CompCert C compiler [BJLM13, BJLM15]. Pierre Roux contributed to Flocq during his postdoctoral stay.

Flocq originated as part of a larger project that aimed at fully verifying the correctness of a simple numerical scheme, by taking into both discretization and round-off errors [BCF+10, BCF+13, BCF+14]. But it appeared that the formalization of real analysis in Coq was not adequate for the discretization part [BLM16]. So, the Coquelicot project was initiated to rethink this formalization and make it more user-friendly [LM12, BLM12, BLM15]. This was the subject of Catherine Lelay's PhD thesis. The conjunction of the polynomial approximations of CoqApprox with the integration theory of Coquelicot then made it possible for CoqInterval to support numerical quadrature [MMSP16, MMSP19].

Finally, collaborating on the Frama-C and Why tools offered a first contact with deductive program verification. This led to the development of the Why3 tool [BFM+13]. Raphaël Rieu-Helft's PhD thesis then aimed at improving Why3, so that it could be used to formally verify a GMP-like multi-precision integer library [RHMM17, MRH18].

Most of my work is related one way or another to the formalization of computer arithmetic. But formalizing computer arithmetic is not sufficient. It is also important to teach the larger public how to use it correctly. This led to writing the Handbook of Floating-point

Arithmetic [MBdD$^+$10, MBdD$^+$18]. Its two editions do not dive much into the topic of formal proofs though, so some lecture notes were later turned into a proper book on Computer Arithmetic and Formal Proofs [BM13, BM17].

To conclude this description of the context of my research, I should mention that I participated to various projects funded by the *Agence Nationale de la Recherche*, either informally, *e.g.*, Cerpan (2005–2008), U3cat (2008–2011), Decert (2008–2011), Tamadi (2010–2013), or as a proper member, *e.g.*, Fost (2008–2011), Verasco (2012–2015), Soprano (2015–2018), Fastrelax (2015–2019).

Chapter 2

Preliminaries

Before going into the thick of the subject, let us see some core concepts. Most of my research work revolves around floating-point arithmetic and its applications, so Section 2.1 gives an overview of some of its mathematical and computational properties. When it comes to automatically verifying theorems, the main foundation is interval arithmetic. Section 2.2 explains how to implement it, why it can be used to formally verify inequalities, and what might cause it to fail to do so. Finally, I hardly ever rely on pen-and-paper proofs, so I make a heavy use of some formal systems to state and prove theorems. These tools, as well as the much needed computational reflection, are presented in Section 2.3.

2.1 Floating-point arithmetic

Floating-point arithmetic has a rather poor reputation. While it behaves as a fast and accurate approximation of real arithmetic most of the time, it sometimes breaks badly. A toy illustration is the lack of associativity of the arithmetic operators, *e.g.*, addition. Given M sufficiently large, the expression $(M \oplus 1) \oplus (-M)$ evaluates to zero, while the expression $(M \oplus (-M)) \oplus 1$ evaluates to 1. As such, one might wonder how floating-point arithmetic could ever be used to perform some *reliable* computations.

The reason is that the IEEE-754 standard gives a concise yet expressive description of floating-point operators [IEE08, §4.3]: "Each operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result [...]." In other words, the standard mandates that floating-point operations behave the same as real operations, except that they must always be followed by a rounding operation. For example, if we denote $\circ$ the rounding operation, the floating-point sum $(M \oplus 1) \oplus (-M)$ can be interpreted as the real expression $\circ(\circ(M + 1) + (-M))$, assuming that some exceptional behaviors did not occur. The precluded behaviors are overflow, *e.g.*, $huge^2 = +\infty$, invalid operation, *e.g.*, $0/0 = \text{NaN}$, and infinitary computations, *e.g.*, $1 + \infty = +\infty$. Computations with floating-point zeros are fine, as long as the sign of zero is never observed, directly or indirectly, *e.g.*, $1/-0 < 0$. Underflowing operations are fine.

In this document, unless explicitly mentioned, floating-point numbers will implicitly be conflated with the real number they represent. This mostly amounts to saying that floating-point inputs will generally be finite and no intermediate computations cause an overflow. Thus, when u and v are real numbers that can be represented by floating-point numbers, $\circ(u + v)$ will represent both a floating-point addition and a real operation followed by a

rounding operation. Operators such as $\oplus$ and $\otimes$ will be used only when exceptional behaviors might occur. When numerous rounding operations occur, the notation $\circ[\ldots]$ will be used. It means that all the inner arithmetic operations are rounded, *e.g.*, $\circ(\circ(M+1)+(-M)) = \circ[(M+1)+(-M)]$.

Section 2.1 gives a bit more details about these rounding operators and their properties. Section 2.1.2 shows how various floating-point formats can be made to fit into a single formalism. Section 2.1.3 tells a few words about what doing effective floating-point computations entails. Finally, Section 2.1.4 shows how the errors introduced by rounding operators can be analyzed.

2.1.1 Rounding operators

Rounding is responsible for the loss of most algebraic properties, but its definition makes it suitable for performing reliable computations. Indeed, given a real number, it returns the floating-point number whose real value is the closest to the input number, according to some rounding direction. For example, rounding toward $+\infty$ for some given floating-point format $\mathcal{F}$ is defined as

$$\triangle(x) = \min\{y \in \mathcal{F} \mid x \leq y\}.$$

By virtue of such a definition, an inequality such as $u+v \leq \triangle(u+v)$ always stands. Thus, we will be able to rely on floating-point arithmetic to deduce properties about real numbers, which explains why floating-point arithmetic can be considered *reliable*.

This raises the question of when $\triangle(x)$ is actually defined, though. The main issue is that $\mathcal{F}$ might not contain sufficiently large floating-point numbers. So, having to handle this corner case would make the verification of floating-point algorithms more tedious than needed. We can avoid this issue by lifting part of the restriction on the exponent range from the standard formats. For example, while the *binary64* format can only represent real numbers in the set

$$\{m \cdot 2^e \mid m, e \in \mathbb{Z} \wedge |m| < 2^{53} \wedge -1074 \leq e \leq 971\},$$

the format will be enlarged as follows, when defining a rounding operator:

$$\{m \cdot 2^e \mid m, e \in \mathbb{Z} \wedge |m| < 2^{53} \wedge -1074 \leq e\}.$$

Note that, unless one of the intermediate real numbers exceeds 2^{1024} in absolute value, the end result of a floating-point computation can be characterized just as well using the format enlarged toward infinities. This means that concerns can be separated when analyzing a floating-point program. On one side, we will look whether exceptional behaviors can occur. On the other side, we will see which real values the program computes when exceptional behaviors are assumed not to occur.

The way the IEEE-754 standard handles overflows and infinitary computations means that, for most floating-point algorithms, we only have to consider the final results to account for exceptional behaviors. For example, as mentioned earlier, a way to perform reliable computations lies in inequalities such as $u+v \leq \triangle(u+v)$. So, if $\triangle(u+v)$ evaluates to $+\infty$, this is still a suitable upper bound on $u+v$, though not as accurate as if the format was actually unbounded. We just have to make sure that these infinite bounds are properly propagated along the computations. Thus, this document will often ignore the issue of overflow, not because there is no overflow, but because it does not have much of a negative impact on the formal verification.

Now that we have made it possible for rounding operators to return arbitrarily large real numbers, let us look at some of their critical properties. First, given a real number x representable in a format $\mathcal{F}$, any rounding $\square(x)$ to this format $\mathcal{F}$ should be x. As a corollary, rounding operators are idempotent:

$$\forall x \in \mathbb{R},\ \square(\square(x)) = \square(x).$$

Another important property is that rounding operators are monotone:

$$\forall x, y \in \mathbb{R},\ x \leq y \Rightarrow \square(x) \leq \square(y).$$

These two properties have numerous consequences. For example, to prove $x < y$ for some real numbers x and y, we just need to check $\square(x) < \square(y)$ by computation. In fact, checking $0 < \square(\square(y) - \square(x))$ works just as well.

2.1.2 Formats

Rounding operators go from $\mathbb{R}$ to a subset of $\mathbb{R}$. Let us characterize these formats, or at least the most useful ones. Given a radix $\beta \in \mathbb{N}$, we define the magnitude of a real number x by $\mathrm{mag}(x) = \lfloor \log_\beta |x| + 1 \rfloor$. In other words,

$$\forall x \neq 0,\ \beta^{\mathrm{mag}(x)-1} \leq |x| < \beta^{\mathrm{mag}(x)}.$$

We will only consider formats that can be represented by an exponent function $\varphi : \mathbb{Z} \to \mathbb{Z}$ in the following way:

$$\mathcal{F} = \{x \in \mathbb{R} \mid x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \in \mathbb{Z}\}.$$

Given a floating-point number x, the integer $\varphi(\mathrm{mag}(x))$ designates its *canonical exponent*, while $x \cdot \beta^{-\varphi(\mathrm{mag}(x))}$ is its *integer significand*. The number $\mathrm{ulp}(x) = \beta^{\varphi(\mathrm{mag}(x))}$ is the unit in the last place of x, or *ulp* for short.

Here are a few useful families of formats to illustrate this definition. A floating-point format with a precision ϱ and an unbounded range of exponents is described by the function $\varphi(e) = e - \varrho$. Such a format, which is part of a family denoted FLX, represents the following real numbers:

$$\{m \cdot \beta^e \mid m, e \in \mathbb{Z} \wedge |m| < \beta^\varrho\}.$$

If we impose a minimal exponent $e_{\min}$, we get the FLT family of formats with *gradual underflow*. If we disregard the issue of overflowing numbers, all the formats from the IEEE-754 standard are part of this family. Table 2.1 describes the values of the parameters for these standard formats. An FLT format is described by the function $\varphi(e) = \max(e - \varrho, e_{\min})$. It represents the following real numbers:

$$\{m \cdot \beta^e \mid m, e \in \mathbb{Z} \wedge |m| < \beta^\varrho \wedge e_{\min} \leq e\}.$$

Finally, fixed-point formats, denoted FIX, are obtained by simply using a constant exponent function $\varphi(e) = e_{\min}$.

Now that formats have gained some structure, we can give a more concrete definition of rounding operators. Indeed, given some way to round a real number r to an integer $\lfloor r \rceil$, the following function is a rounding operator:

$$\square(x) = \lfloor x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rceil \cdot \beta^{\varphi(\mathrm{mag}(x))}.$$

Format	β	ϱ	$e_{\min}$
binary32	2	24	-149
binary64	2	53	-1074
binary128	2	113	-16494
decimal32	10	7	-101
decimal64	10	16	-398
decimal128	10	34	-6176

Table 2.1 – Parameters for the IEEE-754 formats.

As an illustration, here are the definitions of the rounding operators toward $-\infty$ and $+\infty$.

$$\begin{aligned} \bigtriangledown(x) &= \lfloor x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rfloor \cdot \beta^{\varphi(\mathrm{mag}(x))}, \\ \bigtriangleup(x) &= \lceil x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rceil \cdot \beta^{\varphi(\mathrm{mag}(x))}. \end{aligned}$$

Note that some functions of $\mathbb{Z} \to \mathbb{Z}$ are not meaningful as an exponent function φ. More details on the constraints and their rationale can be found in the book [BM17, Ch. 3].

2.1.3 Effective computations

The above definitions of $\bigtriangledown$ and $\bigtriangleup$ are fine for reasoning about rounded computations, but they are kind of unhelpful when it comes to actually performing them. More precisely, they are directly usable only if x happens to be of the shape $m \cdot \beta^e$ with m and e integers.

Thus, for two floating-point numbers $u = m_u \cdot \beta^{e_u}$ and $v = m_v \cdot \beta^{e_v}$, it is easy to represent the exact result of the sum and the product in a shape suitable for rounding:

$$\begin{aligned} u + v &= \left(m_u \cdot \beta^{e_u - \min(e_u, e_v)} + m_v \cdot \beta^{e_v - \min(e_u, e_v)} \right) \cdot \beta^{\min(e_u, e_v)}, \\ u \times v &= (m_u \cdot m_v) \cdot \beta^{e_u + e_v}. \end{aligned}$$

For other operations, the situation is slightly more complicated. Indeed, we cannot represent the exact result as $m \cdot \beta^e$. But we can get an approximation, and for division and square root, we can also get an indication of where the exact result is located with respect to this approximation. This is sufficient to compute a correctly rounded result, as Section 3.2.3 will show. For elementary functions, while we can also compute a floating-point approximation, we have no simple way of knowing where the exact result is located with respect to it. Fortunately, getting correctly rounded elementary functions is not strictly needed to perform reliable computations in a formal setting, as Section 3.3.2 will show.

On the implementation side, I have developed some formal libraries to perform floating-point computations. It started with the Coq library used to check the proofs that Gappa (see Section 4.2.1) generates when verifying fixed- and floating-point algorithms [DM04]. Then came the CoqInterval library (see Section 3.3), which implemented a much more comprehensive set of floating-point operations, though restricted to FLX formats, so as to provide an efficient interval arithmetic [Mel08a]. Eventually, the formalism of both libraries was merged and included into Flocq [BM11]. This Coq formalization was later extended to also support the exceptional behaviors mandated by the IEEE-754 standard (see Section 4.3.2), so that it could be used to specify part of the CompCert C compiler [BJLM13]. All these works on

formalizing floating-point arithmetic were also the occasion to contribute to the 2008 revision of the IEEE-754 standard on floating-point arithmetic [IEE08].

2.1.4 Forward error analysis

When using floating-point arithmetic as a way to approximate computations on real numbers, each rounded operation potentially causes a small error. These errors can vanish during subsequent computations, but more often than not, they tend to persist and grow larger. The final computed result $\tilde{y}$ might thus be quite far from the expected value y, hence the need to analyze how these errors occur and propagate. When looking for bounds on $\tilde{y} - y$, the error is said to be *absolute*, while for $\tilde{y}/y - 1$, it is said to be *relative*.

For the sake of simplicity, let us consider an FLX format, *i.e.*, the precision ϱ is fixed and the exponent range is unbounded, so no underflow can occur. We will look at the various arithmetic operations in turn. First, when rounding a real number to nearest, the absolute and relative errors are bounded as follows:

$$\begin{aligned} |\circ(x) - x| &\leq \frac{1}{2}\mathrm{ulp}(x), \\ \left|\frac{\circ(x) - x}{x}\right| &\leq \frac{1}{2}\beta^{1-\varrho}. \end{aligned}$$

For addition, we can rely on the following implication relating absolute errors. Note that this is a forward analysis: from the error on the inputs, we deduce an error on the result of an operation.

$$|\tilde{u} - u| \leq \delta_u \wedge |\tilde{v} - v| \leq \delta_v \quad \Rightarrow \quad |(\tilde{u} + \tilde{v}) - (u + v)| \leq \delta_u + \delta_v.$$

If the sign of errors might matter (*e.g.*, for error compensation), then the following implication can be used instead:

$$\tilde{u} - u \in \mathbf{\Delta}_u \wedge \tilde{v} - v \in \mathbf{\Delta}_v \quad \Rightarrow \quad (\tilde{u} + \tilde{v}) - (u + v) \in \mathbf{\Delta}_u + \mathbf{\Delta}_v.$$

For multiplication, the errors are as follows:

$$\begin{array}{lllllllll} |\tilde{u} - u| \leq \delta_u & \wedge & |\tilde{v} - v| \leq \delta_v & \Rightarrow & |\tilde{u}\cdot\tilde{v} - u\cdot v| & \leq & \delta_u\cdot|v| + \delta_v\cdot|u| + \delta_u\cdot\delta_v, \\ \left|\dfrac{\tilde{u} - u}{u}\right| \leq \varepsilon_u & \wedge & \left|\dfrac{\tilde{v} - v}{v}\right| \leq \varepsilon_v & \Rightarrow & \left|\dfrac{\tilde{u}\cdot\tilde{v} - u\cdot v}{u\cdot v}\right| & \leq & \varepsilon_u + \varepsilon_v + \varepsilon_u\cdot\varepsilon_v. \end{array}$$

Note that the product $\varepsilon_u \cdot \varepsilon_v$ is usually negligible with respect to the other terms. So, when doing back-of-the-envelope computations, a first-order approach can be used. This gives the following rule of thumb: "the relative error of the product is the sum of the relative errors". This is similar for the division, or rather, using enclosures: "the relative error of the division is the difference of the relative errors". Finally, the rule of thumb for the square root is that "it halves the relative error".

Note that we have assumed that the last computation was performed exactly. In practice, it will be a rounded computation, so we need some way to compose errors. The composition formula for the absolute error is just the triangular inequality. For the composition of relative errors, the formula is similar to the one for the product:

$$\left|\frac{u - v}{v}\right| \leq \varepsilon_1 \wedge \left|\frac{v - w}{w}\right| \leq \varepsilon_2 \quad \Rightarrow \quad \left|\frac{u - w}{w}\right| \leq \varepsilon_1 + \varepsilon_2 + \varepsilon_1 \cdot \varepsilon_2.$$

As in the case of the multiplication, the $\varepsilon_1 \cdot \varepsilon_2$ term can be neglected when composing errors at first order. As a consequence, the first-order formula for the rounded product is

$$\left|\frac{\circ(\tilde{u} \cdot \tilde{v}) - u \cdot v}{u \cdot v}\right| \lesssim \varepsilon_u + \varepsilon_v + \frac{1}{2}\beta^{1-\varrho}.$$

Notice that this section has explained how relative errors are propagated along products, but not along sums. The reason is that there is no simple rule. In fact, a cancellation might well cause relative errors to explode. The way the Gappa tool tackles this problem will be presented in Section 4.2.2.

2.2 Interval arithmetic

Floating-point arithmetic has sufficiently many good properties to obtain reliable results regarding real numbers, but forward error analysis makes for complicated proofs. So, unless performance mandates it, it is much easier to build a dedicated arithmetic on top of floating-point arithmetic. Instead of approximating a real number by a floating-point number, we will enclose it by two floating-point numbers. This is one possible implementation of *interval arithmetic* [Moo63].

Fundamentally, interval arithmetic is about reasoning about sets of real numbers, as shown in Section 2.2.1. The advantage of these sets is that they are easy to compute, as shown in Section 2.2.2. Unfortunately, the computed sets might be so large that nothing interesting can be deduced from them, as shown in Section 2.2.3.

2.2.1 Enclosures

Let $\mathbb{I}$ denote the set of intervals, *i.e.*, closed connected subset of real numbers. An *interval extension* of a real-valued function $f : \mathbb{R}^n \to \mathbb{R}$ is a function $\mathbf{f} : \mathbb{I}^n \to \mathbb{I}$ that satisfies the *containment* property:

$$\forall \mathbf{x}_1 \in \mathbb{I}, \ldots, \mathbf{x}_n \in \mathbb{I},\ \forall x_1 \in \mathbb{R}, \ldots, x_n \in \mathbb{R},$$
$$x_1 \in \mathbf{x}_1 \wedge \ldots \wedge x_n \in \mathbf{x}_n \Rightarrow f(x_1, \ldots, x_n) \in \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n).$$

The containment property opens the way to proving some properties on real-valued expressions. For example, in order to to prove $\forall x \in \mathbf{x},\ f(x) \geq A$, it is sufficient to exhibit an interval extension $\mathbf{f}$ of f such that $\mathbf{f}(\mathbf{x}) \subseteq [A; +\infty)$.

Note that the interval extension $\mathbf{f}$ is not uniquely defined. Given some interval extension $\mathbf{f}_1$ of f, any function $\mathbf{f}_2$ that satisfies the following property will also be an interval extension:

$$\forall \mathbf{x}_1 \in \mathbb{I}, \ldots, \mathbf{x}_n \in \mathbb{I},\ \mathbf{f}_1(\mathbf{x}_1, \ldots, \mathbf{x}_n) \subseteq \mathbf{f}_2(\mathbf{x}_1, \ldots, \mathbf{x}_n).$$

Interval extension $\mathbf{f}_1$ is said to be *tighter* than $\mathbf{f}_2$.

The *set extension* of f is the function of $\mathbb{I}^n \to \mathcal{P}(\mathbb{R})$ defined as follows:

$$f(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \{f(x_1, \ldots, x_n) \mid x_1 \in \mathbf{x}_1 \wedge \ldots \wedge x_n \in \mathbf{x}_n\}.$$

Since $f(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ is generally not a closed connected subset of $\mathbb{R}$, the set extension of f is not a proper interval extension. It can, however, be turned into the tightest interval extension by computing the closed convex hull:

$$\mathbf{f}_{\text{tightest}}(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \text{hull}(f(\mathbf{x}_1, \ldots, \mathbf{x}_n)).$$

On the contrary, the least useful interval extension returns the widest possible interval: for any input, we have

$$\mathbf{f}_{\text{worst}}(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \mathbb{R}.$$

In practice, we can get arbitrarily close to the tightest extension, but usually at the expense of an increase of computation time.

2.2.2 Effective computations

There are several ways to represent an interval $\mathbf{u}$, but all the tools and libraries I have developed use an *inf-sup* representation: $[\underline{u}; \overline{u}]$. The bounds are meant to be real numbers (in practice, finite floating-point numbers), but in order to accommodate half-bounded intervals or $\mathbb{R}$, we might also allow $\underline{u} = -\infty$ and $\overline{u} = +\infty$.

Let us now define some tight interval extensions of the basic arithmetic operators. The enclosures below are easily obtained by remembering that addition, subtraction, and multiplication are continuous functions with some monotony properties. Moreover, we are using directed rounding. Given two enclosures $u \in [\underline{u}; \overline{u}]$ and $v \in [\underline{v}; \overline{v}]$ with finite bounds, we have

$$\begin{aligned}
u + v &\in [\bigtriangledown(\underline{u} + \underline{v}); \bigtriangleup(\overline{u} + \overline{v})], \\
u - v &\in [\bigtriangledown(\underline{u} - \overline{v}); \bigtriangleup(\overline{u} - \underline{v})], \\
u \cdot v &\in [\min(\bigtriangledown(\underline{u} \cdot \underline{v}), \bigtriangledown(\underline{u} \cdot \overline{v}), \bigtriangledown(\overline{u} \cdot \underline{v}), \bigtriangledown(\overline{u} \cdot \overline{v})); \\
&\quad\ \max(\bigtriangleup(\underline{u} \cdot \underline{v}), \bigtriangleup(\underline{u} \cdot \overline{v}), \bigtriangleup(\overline{u} \cdot \underline{v}), \bigtriangleup(\overline{u} \cdot \overline{v}))].
\end{aligned}$$

Note that the formulas above are meaningful only for finite bounds. In the case of addition and subtraction, they are easily extended to infinite bounds, since the undefined operation $\infty - \infty$ cannot occur. For multiplication, any occurrence of the product $0 \cdot \infty$ in the last formula should be evaluated as zero (which is not what IEEE-754 mandates, unfortunately).

Once we have interval extensions for all the basic arithmetic operators, computing enclosures of more complicated expressions is just a matter of composing these extensions. As an illustration, let us prove $u \cdot (v + \sqrt{v}) \geq -13$ for $u \in [-3; 4]$ and $v \in [1; 2]$. Using a decimal floating-point arithmetic with a precision of two digits, we get

$$\begin{aligned}
u \cdot (v + \sqrt{v}) &\in [-3; 4] \cdot ([1; 2] + \sqrt{[1; 2]}) \\
&\in [-3; 4] \cdot ([1; 2] + [1; 1.5]) \\
&\in [-3; 4] \cdot [2; 3.5] \\
&\in [-11; 14].
\end{aligned}$$

Since $-11 \geq -13$, this concludes the proof. Note that, if we had used only one digit of precision, we would have obtained $[-20; 20]$ as the final interval, which would not be sufficiently tight to prove $u \cdot (v + \sqrt{v}) \geq -13$.

Notice that, for basic arithmetic operators, we do not visually distinguish the real operator from its interval extensions. So, whenever an arithmetic operator takes interval inputs, it should be understood as any interval extension of the corresponding operator on real numbers. Moreover, whenever a real number appears as an input of an interval operator, it should be understood as any interval that encloses this number. For example, the expression $(v - u) \cdot \mathbf{x}$ denotes the interval product of the interval $\mathbf{x}$ with any (hopefully tight) interval enclosing the real $v - u$.

Libraries and standard

I have developed two libraries offering interval arithmetic. The first one is the Boost.Interval library. It is a C++ library that was developed under the supervision of Hervé Brönnimann and Sylvain Pion [BMP03, BMP06b]. It is part of the Boost project.[1] Contrarily to other libraries that existed at the time, *e.g.*, Profil/BIAS [Knü94], the focus was put on genericity. Indeed, the library is not restricted to using hardware floating-point numbers as interval bounds. It also supports rational numbers, multi-precision floating-point numbers, and so on.

The other library is CoqInterval, a formalization of interval arithmetic for the Coq proof assistant, which will be described in Section 3.3. It provides interval-based automated procedures for formally verifying enclosures of real-valued expressions [Mel08b]. To do so, it formalizes floating-point arithmetic in an effective way [Mel08a, Mel12]. Initially, the library computed interval extensions by composing interval operators and by using automatic differentiation. Later, polynomial approximations were formalized to provide tighter interval extensions [MDM16]. Finally, CoqInterval was also extended with support for numerical integration [MMSP16, MMSP19]. More details on these features will be given in Sections 3.4 and 3.5, respectively.

As both a user and designer of interval arithmetic libraries, I also got involved in the IEEE-1788 standard on interval arithmetic [EM09, IEE15]. Unsurprisingly, it standardizes a representation of intervals with floating-point bounds, as well as some interval operators, which have to satisfy the containment property.

The design of the Boost.Interval library was also the occasion to interact with the standardization committee for the C++ language through various proposals to improve the language for reliable numerical computations [BMP06a, BMP06c, BMP06d, MP09]. None of them made it into the actual standard, but they at least raised the awareness of the language designers on this topic. Now that there is an international standard for interval arithmetic, as there was for floating-point arithmetic, it might be worth reviving the effort of getting interval arithmetic inside the C++ standard.

2.2.3 Dependency effect

As shown by the example of $u \cdot (v + \sqrt{v})$ above, if the floating-point arithmetic used when computing the bounds is not precise enough, interval arithmetic might fail to compute tight enough bounds on an expression.

There is another common case of failure of interval arithmetic. When composing the interval extensions, the relation between multiple occurrences of a given subexpression is lost. This is the *dependency effect*. Let us illustrate this effect. For all $x \in [0;1]$, we wish to bound the signed distance $\tilde{y} - y$ between an ideal value $y = \exp(x)$ and its approximation $\tilde{y} = 1 + x + x^2/2$. Using an infinitely precise arithmetic (except for the very last interval), this would give the following enclosure:

$$\begin{aligned} \tilde{y} - y &\in (1 + \mathbf{x} + \mathbf{x}^2/2) - \exp(\mathbf{x}) \\ &\in ([1;1] + [0;1] + [0;0.5]) - [\exp 0; \exp 1] \\ &\in [1;2.5] - [1;e] \\ &\in [1-e; 2.5-1] \subseteq [-1.72; 1.5]. \end{aligned}$$

[1] `http://www.boost.org/`

So, the interval evaluation makes it possible to automatically conclude that $|\tilde{y} - y| \leq 1.72$ for $x \in [0; 1]$. While correct, this upper bound is a gross overestimation of the supremum of $|\tilde{y} - y|$. Indeed, $\tilde{y} - y$ is equal to $\sum_{k \geq 3} x^k / k!$, which is an increasing function of $x \in [0; 1]$. So, the extrema of $\tilde{y} - y$ are reached for $x = 0$ and $x = 1$, which makes it possible to manually prove that $|\tilde{y} - y| \leq 0.22$. Since the computed upper bound 1.72 is so far from the actual supremum 0.22, it is hardly useful to prove anything interesting.

The reason for the overestimation comes from the multiple occurrences of the input variable x in the difference $\tilde{y} - y$. Interval arithmetic happens to give tight enclosures of $y \in [1; e]$ and $\tilde{y} \in [1; 2.5]$, but these two enclosures do not carry enough information to compute a tight enclosure of $\tilde{y} - y$.

All is not lost, though. Interval implementations usually enjoy an *isotony* property, that is, smaller input intervals lead to smaller output intervals. Ultimately, if the input intervals are point intervals, the output intervals will be point intervals too (disregarding round-off errors). So, using smaller input intervals tends to reduce the dependency effect. The simplest way to reduce this effect is thus to subdivide the input intervals into subintervals by way of *bisection*. For example, instead of using $\mathbf{x} = [0; 1]$ to bound $\tilde{y} - y$, we can separately consider $\mathbf{x} = [0; 0.5]$ and $\mathbf{x} = [0.5; 1]$, which gives

$$\begin{aligned} \tilde{y} - y &\in [-0.65; 0.63] \quad \text{when} \quad x \in [0; 0.5], \\ \tilde{y} - y &\in [-1.10; 0.86] \quad \text{when} \quad x \in [0.5; 1]. \end{aligned}$$

Then we merge the results by taking the union of the two enclosures:

$$\tilde{y} - y \in [-1.10; 0.86] \quad \text{when} \quad x \in [0; 1].$$

As expected, this new enclosure is tighter than the one obtained by interval evaluation on $[0; 1]$. If one wants to prove that $|\tilde{y} - y| \leq 0.65$, then the enclosure obtained for $x \in [0; 0.5]$ is sufficient. The one obtained for $x \in [0.5; 1]$ is not; it should be refined further by considering subintervals of $[0.5; 1]$.

Oracles

Performing interval operations in the context of a formal system might be several orders of magnitude slower than in a dedicated program. For example, it is not unheard of for the Coq-Interval library (see Section 3.3) to be 1000 times slower than a program using Boost.Interval with hardware floating-point numbers. As a consequence, it might be important to perform only the interval computations that are strictly necessary.

Consider the inequality $x + x^{-1} \geq 1.6$ for $x \in [1; 1025]$. The left-hand-side expression is increasing over the input interval, but let us ignore this property and turn to naive interval arithmetic to bound the expression. Performing an interval computation with $[1; 1025]$ is not sufficient to verify the property, so we need to subdivide the input domain. To do so, we perform a bisection until the property is verified on each subintervals. We end up considering $[1; 1025]$, $[513; 1025]$, $[1; 513]$, $[257; 513]$, $[1; 257]$, and so on until $[1.5; 2]$ and $[1; 1.5]$. In other words, we have to compute 23 enclosures of $x + x^{-1}$ in order to verify that it is larger than 1.6. Yet, only two enclosures are actually needed: one for $[1; 1.6]$ and the other one for $[1.6; 1025]$.

Since performing computations outside a formal system is so much cheaper, it might be worth implementing an oracle that will look for an optimal splitting of the input domain. Even if the splitting is suboptimal, *e.g.*, $[1; 1.5] \cup [1.5; 2] \cup [2; 1025]$ for the example above,

it will still be worthwhile. Note that bug-ridden oracles do not endanger the soundness of the formal system, since we can (and shall) easily and formally check that the splitting they produce is indeed a covering of the input domain. We can even make it so that they do not endanger the relative completeness of the system either. Indeed, if a bisection of the original input domain would lead to a formal proof of the property, a bisection of an oracle-computed splitting would too.

Marc Daumas, César Muñoz, and I, used this oracle-based approach in order to formally verify enclosures using the PVS proof system [DMM05]. A C++ oracle built on top of Boost.Interval was looking for the best way to split the input domain. Since the PVS interval library was using truncated power series to enclose functions, the oracle was also trying to minimize the number of power series terms that were actually needed to verify a given enclosure. It should be noted that, in that work, the dependency effect was reduced not just by using bisection, but also by using first-order Taylor expansions. This method as well as related ones will be described in more details in Section 3.4.2.

Decidability

Since we just talked about soundness and completeness, we might just as well talk about *decidability*. An algorithm decides a property P if it answers "yes" when P holds, and "no" otherwise. An algorithm semi-decides a property P if it answers "yes" when P holds, and does not answer "yes" otherwise (*i.e.*, either it answers "no" or it does not terminate). Neither of these two definitions apply here, since equality on real numbers becomes undecidable when we start supporting more than just basic arithmetic operators.

Interval arithmetic, however, enjoys δ-decidability [GAC12]. An algorithm δ-decides a property P if it gives the correct answer whenever there exists a neighborhood of P that only contains properties P' that agree with P. Otherwise, the algorithm either answers correctly or does not terminate.

Let us skip the actual definition of neighborhood and let us illustrate it on an example. The property P we want to prove is $\forall x \in [0;1],\ x - x \leq 0.1$. A property $P'_{\vec{\delta}}$ close to it is obtained by adding some noise symbols on the inputs, constants, and intermediate results:

$$\forall x \in [0;1],\ (x + \delta_1) - (x + \delta_2) + \delta_3 \leq 0.1 + \delta_4.$$

Property P holds, and so does $P'_{\vec{\delta}}$ when $\|\vec{\delta}\|_1 = |\delta_1| + \cdots + |\delta_4| \leq 0.1$. So, there exists a neighborhood of P that agrees with P. As such, naive interval arithmetic, combined with interval subdivisions and precision increase, will succeed in proving that P holds. More precisely, splitting the input interval in more that 20 subintervals should succeed in proving P by interval computations.

If the consequent of property P had been $x - x \leq 0$ instead, then there would be no neighborhood of P where all the properties agree with P. As such, an algorithm based on interval arithmetic might not succeed in proving P. In fact, even by subdividing the input interval and increasing precision, naive interval arithmetic will never succeed.

Notice that we have talked only about decidability and not about practical effectiveness. While the above approach can theoretically succeed in proving any consequent $x - x \leq \varepsilon$ for $\varepsilon > 0$, the number of subdivisions needed (and thus the time it takes) grows as fast as $1/\varepsilon$, hence the adjective *naive*. So, most of the work on interval arithmetic revolves around devising approaches that defeat the dependency effect. Some of these methods will be detailed in Section 3.4.2.

2.3 Formal verification

Logical reasoning aims at checking every step of the proof, so as to guarantee only justified assumptions and 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 shape 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 that these syntactic rules are correctly applied, a mechanical device as stupid as a computer can be used. It will be faster and more systematic than a human being, as long as the proof is comprehensive and given in a language understandable by the computer. That does not mean that the computer is a perfect proof checker. For example, the inference rules might be flawed and allow meaningless deduction. Even if they are correct, the program tasked with verifying their correct application might contain bugs. Moreover, the environment might also interfere with the verification process, be it due to processor flaws, operating system bugs, or cosmic rays. Still, despite all these potential issues, computers seem like a better fit for carefully verifying proofs, since they are much less likely to have their concentration wane that a human being.

Section 2.3.1 presents three formal systems that were used to prove theorems and verify programs: Coq, Alt-Ergo, and Why3. Section 2.3.2 presents a proof method that I intensively use when these systems fall short: reflection.

2.3.1 Formal systems

Three tools will be regularly mentioned throughout this document. Their primary purpose is to let users state theorems and to verify that they hold. Depending on the tools, the user might have to provide a detailed proof of these theorems to guide the tools or the tools might automatically look for proofs.

Coq

The first tool is the Coq proof assistant,[2] which provides a higher-order logic with dependent types [BC04]. The fundamental property of the system is that mathematical statements are interpreted as function types. For example, an implication $A \Rightarrow B$ can be understood as the type of a program that transforms a proof of A into a proof of B. If such a program exists and terminates for any proof of A, then the implication holds. Thus, the process of verifying a theorem in Coq consists in exhibiting a program. Coq automatically checks that this program is well-typed and is terminating. So, at its core, Coq is nothing more than a typed programming language and the associated type checker.

For example, Figure 2.1 shows a short Coq script containing three definitions. The first one *inductively* defines the set of nonnegative integers as being generated by a nullary symbol O (zero) and a unary symbol S (successor). The second one defines the addition as a recursive function that implements the two rules $O + y = y$ and $S(x') = x' + S(y)$ by pattern-matching. Finally, the third definition is a recursive function that takes an integer x and returns a proof of $\forall y,\ S(x) + y = S(x + y)$.[3] Thus, this last definition is a proof of $\forall x\, y,\ S(x) + y = S(x + y)$.

[2] `https://coq.inria.fr/`

[3] The fact that this function is actually well-typed is not trivial.

```
Inductive nat : Set := O | S : nat -> nat.
Fixpoint plus (x y : nat) : nat :=
  match x with
  | O => y
  | S x' => plus x' (S y)
  end.
Fixpoint plus_S (x y : nat) : plus (S x) y = S (plus x y) :=
  match x with
  | O => eq_refl
  | S x' => plus_S x' (S y)
  end.
```

Figure 2.1 – Peano integers and the proof of $\forall x\, y,\ S(x) + y = S(x + y)$.

```
Lemma plus_S : forall x y, plus (S x) y = S (plus x y).
Proof.
induction x as [|x' IH].
- easy.
- intros y.
  apply (IH (S y)).
Qed.
```

Figure 2.2 – A tactic-based proof of $\forall x\, y,\ S(x) + y = S(x + y)$.

In the example above, the user has explicitly built a term whose type matches this simple theorem. But for more complicated ones, this approach is no longer practical. So, Coq proposes a way to interactively build terms by using *tactics*. At any point of the proof process, the user is presented with several goals that represent the missing pieces of a proof. A goal is split into a *context* containing several variables and hypotheses, and a conclusion that has to be proved given these variables and hypotheses. (From the perspective of building functions, these variables and hypotheses are the arguments of a function and the conclusion is the return type of the function.) Initially, there is only one goal, which represents the theorem to be proved. Successfully applying a tactic replaces the current goal by some new goals (possibly none). Once there is no goal left, the theorem has been successfully proved.

The Coq script on Figure 2.2 illustrates how one can obtain a proof of $\forall x\, y,\ S(x) + y = S(x + y)$ using tactics. After having typed `induction x`, the user is presented with two goals. The conclusion of the first one is $\forall y,\ S(O) + y = S(O + y)$. The conclusion of the second one is $\forall y,\ S(S(x')) + y = S(S(x') + y)$ and the context contains a variable x' and a hypothesis `IH` that states $\forall y,\ S(x') + y = S(x' + y)$. The user can then tackle both new goals in turns, using just the `easy` tactic for the first one, and two tactics for the second one.

Alt-Ergo

The second tool is Alt-Ergo,[4] a theorem prover based on *satisfiability modulo theory* or SMT for short [Con12]. Contrarily to Coq, this is an automated prover. The user only states propositions and the tool tries to automatically prove their validity. The user has no way to indicate a proof that would be verified by the tool. The input language of Alt-Ergo is a first-order logic with some built-in theories and polymorphic data types. Alt-Ergo's input language is especially well suited for verification conditions generated by Why3 (see below).

Given a set of universally quantified lemmas and a proposition, Alt-Ergo tries to deduce the proposition from the lemmas and its built-in theories. To do so, it tries to prove that the negation of the proposition is unsatisfiable. The main loop starts with a SAT solver, which takes care of disjunctions and produces conjunctions of literals. All these conjunctions have to be unstatisfiable. The relevant parts of a conjunction are sent to the solvers of the built-in theories. The difficult part lies in properly combining the answers of these solvers, since a conjunction might be seen as satisfiable by each solver in isolation, but might be unsatisfiable as a whole, *e.g.*, due to inter-theory equalities. If a conjunction is still believed to be satisfiable at this point, lemmas are instantiated with ground terms to create new literals and the process starts again. Instances are selected by a *trigger* mechanism, which is succinctly described, as well as a few improvements, in Section 4.5.2.

The expressiveness of SMT solvers lies in the built-in theories they support. In addition to equality and uninterpreted symbols, the most important ones in the context of program verification are the theories of functional arrays, integer arithmetic, and rational arithmetic. Some improvements to Alt-Ergo's solver for integer arithmetic are described in Section 5.1.1. To further extend Alt-Ergo, a built-in theory of floating-point arithmetic was added, whose solver is described in Section 4.5.2.

Why3

The third tool is Why3,[5] a platform for deductive program verification [BFMP11, FP13]. It provides a rich language, called WhyML, to write programs and their logical specifications. It relies on external theorem provers, automated (Alt-Ergo, CVC4, Eprover, Z3, and so on) and interactive (Coq, Isabelle, PVS), to discharge verification conditions. It was mostly developed by François Bobot, Jean-Christophe Filliâtre, Claude Marché, Andrei Paskevich, and me.

Why3 is based on first-order logic with rank-1 polymorphic types, algebraic data types, inductive predicates, and several other extensions. Its programming language, WhyML, is inspired by ML and provides inductive datatypes, pattern matching, mutability, exceptions, automatic memory reclamation, anonymous functions, and so on. We have designed WhyML so that it is comfortable to use as a primary programming language. But it can also be used as an intermediate language for the verification of C/ACSL, Java/JML, and Ada/SPARK annotated programs [FM07, MK15].

Figure 2.3 adapts the example of Figure 2.1 to Why3. First, it defines the inductive type of Peano's integers. Then, it defines recursively the addition of two integers. Finally, it states and proves $\forall x\, y,\ S(x) + y = S(x + y)$.

The `let rec` keywords start the recursive definition of a WhyML program. The `function` keyword indicates that this definition should be made accessible from Why3's logic. The `lemma`

[4] `https://alt-ergo.ocamlpro.com/`
[5] `http://why3.lri.fr/`

```
type nat = O | S nat

let rec function plus (x y: nat): nat =
  match x with
  | O -> y
  | S x' -> plus x' (S y)
  end

let rec lemma plus_S (x y: nat): unit
  ensures { plus (S x) y = S (plus x y) }
=
  match x with
  | O -> ()
  | S x' -> plus_S x' (S y)
  end
```

Figure 2.3 – Peano integers (Why3 version).

keyword indicates that the contract of the function (here, only a postcondition, introduced by `ensures`) should be turned into a lemma. In other words, this long definition of `plus_S` has the same effect as the following declaration, ultimately.

```
lemma plus_S: forall x y: nat. plus (S x) y = S (plus x y)
```

Unfortunately, a theorem prover such as Alt-Ergo struggles to automatically verify propositions by induction. Thus, by giving a body to `plus_S`, the user guides the solver toward the actual proof. Indeed, the verification condition sent to the solver is no longer just

$$\forall x\, y,\ S(x) + y = S(x + y).$$

Instead, Why3 computes the weakest precondition that guarantees that the function satisfies its postcondition [Dij75]. In the case $x = O$, the verification condition is exactly the postcondition, since the function does not do anything. In the case $x = S(x')$, the postcondition of `plus_S` should be implied by the postcondition of its recursive call. So, the complete verification condition is

$$\forall x\, y\, x',\ (x = O \Rightarrow S(x) + y = S(x + y)) \wedge (x = S(x') \Rightarrow S(x') + S(y) = S(x' + S(y)) \Rightarrow S(x) + y = S(x + y)).$$

This new verification condition no longer needs any induction to be verified. Unfolding the definition of addition and doing a bit of equational reasoning are sufficient, so Alt-Ergo can now discharge the goal instantly. As far as I am concerned, the idea of reusing programs as a way to prove lemmas was mostly inspired by Leino's work [Lei13], though my experience with Coq had put me in the right mindset.

The example above illustrates the use of Why3 to express theories and verify them. But it shows few features of WhyML as a programming language. So, Figure 2.4 presents a

```
let plus' (x y: nat): nat
  ensures { result = plus x y }
=
  let u = ref x in
  let v = ref y in
  while true do
    invariant { plus !u !v = plus x y }
    variant { !u }
    match !u with
    | O -> break
    | S u' -> u := u'; v := S !v
    end
  done;
  !v
```

Figure 2.4 – Derecursified Peano addition.

derecursified imperative version of Peano's addition. Its specification states that the function returns the same result as the original function.

As in OCaml, the `ref` function creates a fresh memory cell (initialized with its argument), which can be accessed using `!` and can be modified using `:=`. Loops can be annotated with invariants and variants. A `variant` describes a quantity that decreases with respect to some well-founded ordering; this guarantees the termination of the loop. An `invariant` states a property that is true at the entrance of the loop and is preserved at each iteration. The generated verification condition states that those are indeed the variant and invariant of the loop. Once a function has been verified using Why3, it can be extracted to another programming language such as OCaml.

There are a few features of Why3 that are hardly found in other software verification frameworks. First of all, Why3 does not target a single automated solver but as many of them as possible. Depending on the verification condition, the user can then decide to use one or another solver (or several of them to increase confidence). The user might also decide to not directly call a solver on a verification condition, but to first transform it into several new verification conditions, hopefully simpler for the solvers. To preserve the user decisions regarding provers and transformations over time, Why3 offers a mechanism of persistent session. In particular, when the user modifies the code or the specification, Why3 tries its best to adapt the old decisions to the new verification conditions [BFM+13].

Why3 also supports interactive theorem provers such as Coq. There is not much difference with automated solvers, except that it is now up to the user to prove the verification conditions. But interactive theorem provers are not used just for verification conditions generated from programs. Indeed, Why3 comes with a standard library of functions, theorems, and data structures. Any function with a specification but no body should be considered as an axiom. To increase the confidence in the consistency of this library, I have realized part of them. That means that, for almost every such function declaration, I have used Coq to verify that there actually exists some corresponding definition and that it is consistent with

any theorem mentioning it. Section 4.5.2 will show what the realization for the theory of floating-point arithmetic looks like.

2.3.2 Computational reflection

When using formal systems, it is not just a matter of stating theorems or specifying programs, it is also a matter of verifying them. That is where the user might be hitting a wall. If the primitives offered by the systems (tactics for Coq, external automated provers for Why3) are not powerful enough for the problem at hand, the proofs become verbose and tedious. It is then time to extend the theorem prover with new capabilities, *e.g.*, an inference rule dedicated to the current proof. As Harrison states it, a way to do so is to "incorporate a reflection principle, so that the user can verify within the existing theorem proving infrastructure that the code implementing a new rule is correct, and to add that code to the system" [Har95].

To design a reflection-based decision procedure for a family $\mathcal{P}$ of propositions, one first embeds these propositions into the logical language of the formal system. Given $P \in \mathcal{P}$, let us denote $\ulcorner P \urcorner$ its embedding, *e.g.*, the abstract syntax tree of P. Then, one devises a predicate ψ such that, for any $P \in \mathcal{P}$, if $\psi(\ulcorner P \urcorner)$ holds, then P holds. If the implication $\psi(\ulcorner P \urcorner) \Rightarrow P$ has been formally verified, then to obtain a proof of P, one can instead search for a proof of $\psi(\ulcorner P \urcorner)$, hence the name *reflection*. Moreover, if ψ is such that $\psi(\ulcorner P \urcorner)$ can be proved just by performing computations in the logical system, then we have a proof procedure by *computational reflection* [Bou97, GM05, Bes07, CN08, CGP17].

Let us illustrate this process on a toy example: the correctness of Strassen's matrix multiplication algorithm. Among other properties, one has to prove four matrix equalities such as the following one:

$$A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} = M_{1,1},$$

with

$$\begin{aligned} M_{1,1} &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) + A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\ &- (A_{1,1} + A_{1,2}) \cdot B_{2,2} + (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}). \end{aligned}$$

By the group laws of matrix addition and by distributivity of matrix multiplication, one easily shows that the right-hand side of the equality can be turned into the left-hand side. Unfortunately, in a formal setting, it means instantiating these algebraic laws on the order of one hundred times per matrix equality, assuming they are applied in an optimal way. That makes it an exhausting proof for Coq users, while Why3 users would see their favorite automated solvers give up (*e.g.*, Alt-Ergo, CVC4, Eprover, Z3). Thus, without a decision procedure dedicated to equalities over a non-commutative ring, the proof might stay out of reach.

Let us see how to supplement the shortcomings of these systems using Why3. Note that there is nothing specific to Why3. The exact same process could be followed in any formal system offering some computational capabilities.

The first step is to embed both sides of the equality into the logical language of Why3. We define the following inductive type `t` to represent its abstract syntax tree:

```
type t = Var int | Add t t | Mul t t | Sub t t
```

Matrices appearing at the leaves of the expression, *e.g.*, $A_{2,1}$, are assigned a unique integer identifier and are represented using the `Var` constructor. The sum, product, and differences of two matrices, are represented using the constructors `Add`, `Mul`, and `Sub`.

Note that the function $M \mapsto \ulcorner M \urcorner$ cannot be expressed in the logical language, but its inverse can. We thus define a function that maps a term of type `t` into a matrix, as shown below. This definition causes Why3 to create a recursive function `interp` inside the logical system, since its termination is visibly guaranteed by the structural decrease of its argument `x`.

```
type vars = int -> a
let rec function interp (x: t) (y: vars) : a =
  match x with
  | Var n -> y n
  | Add x1 x2 -> aplus  (interp x1 y) (interp x2 y)
  | Mul x1 x2 -> atimes (interp x1 y) (interp x2 y)
  | Sub x1 x2 -> asub   (interp x1 y) (interp x2 y)
  end
```

When `aplus`, resp. `atimes`, is instantiated using matrix sum, resp. product, one can prove that the Why3 term `(interp (Mul (Add (Var 0) (Var 1)) (Var 7)) y)` is equal to $(A_{1,1} + A_{1,2}) \cdot B_{2,2}$, assuming that `y` maps 0 to $A_{1,1}$, 1 to $A_{1,2}$, and 7 to $B_{2,2}$. This proof can be done by unfolding the definition of `interp`, by reducing the `match with` constructs, and by substituting the applications of `y` by the corresponding results. Why3 provides a small rewriting engine that is powerful enough for such a proof, but one could also use an external prover.

Let us suppose that we now have two concrete expressions `x1` and `x2` of type `t` and a single map `y` of type `vars` and that we want to prove the following equality:

```
lemma g: interp x1 y = interp x2 y
```

The actual value of `y` does not matter, but the facts that `aplus` is a group operation and that `amult` is distributive do. In other words, we want to see `x1` and `x2` as non-commutative polynomials and we want to prove that they have the same monomials with the same coefficients. To do so, let us turn them into weighted lists of monomials using the `conv` function. An excerpt of its code and specification is given in Figure 2.5. For example, the term $(A_{1,1} + A_{1,2}) \cdot B_{2,2}$ gets turned into the list

```
Cons (M 1 (Cons 0 (Cons 7 Nil))) (Cons (M 1 (Cons 1 (Cons 7 Nil))) Nil)
```

Note that we have introduced a new interpretation function `interp'` and we have stated the postcondition of `conv` accordingly. Why3 requires us to prove that this postcondition holds. The proof is straightforward, even in the multiplication case. Once done, we obtain the following lemma in the context:

```
lemma conv_def: forall x y. interp x y = interp' (conv x) y
```

We define one last function, `norm`, which sorts a weighted list of monomials by insertion using a lexicographic order, merging contiguous monomials along the way. Its postcondition, once proved, leads to another lemma in the context:

```
lemma norm_def: forall x y. interp' x y = interp' (norm x) y
```

Note that we do not even need to prove that `norm` actually sorts the input list or that it merges monomials, so the proof is again trivial. If there is some bug in `norm`, it only endangers the completeness of the approach, not its soundness. For example, defining `norm` as the identity function would ultimately be fine but pointless.

```
type m = M int (list int)
type t' = list m

let rec function interp' (x: t') (y: vars) : a =
  match x with
  | Nil -> azero
  | Cons (M r m) l -> aplus (($) r (mon m y)) (interp' l y)
  end

let rec function conv (x:t) : t'
  ensures { forall y. interp x y = interp' result y }
= match x with
  | Var v -> Cons (M rone (Cons v Nil)) Nil
  | Add x1 x2 -> (conv x1) ++ (conv x2)
  | Mul x1 x2 -> ...
  end
```

Figure 2.5 – Converting a polynomial to a list of monomials.

By composing `norm` and `conv` and equality, we get our decision procedure ψ dedicated to verifying Strassen's algorithm. Indeed, to prove the goal `g` above, we just need to prove the following intermediate lemma:

```
lemma g_aux: norm (conv x1) = norm (conv x2)
```

As with `interp` before, `norm` and `conv` are logical functions defined by induction on their argument, so there is no difficulty in proving `g_aux` using the rewriting engine of Why3 or an external automated prover.

A very similar reflection-based tactic is used by the Coq proof assistant to formally verify equalities in a commutative ring or semi-ring [GM05]. This tactic was implemented, part as an OCaml plugin for Coq, part in the meta-language Ltac of Coq. Rather than lists of monomials, that work uses Horner's representation of polynomials: $p_0 + x_1 \cdot (p_1 + x_1 \cdot (p_2 + x_1 \cdots))$ with $(p_i)_i$ being polynomials where variable x_1 does not occur.

One important property of defining a decision procedure by reflection is that the soundness of the system is not endangered, since the user has to prove the correctness of the procedure (*e.g.*, the lemmas `conv_def` and `norm_def`).

We have not yet explained how one obtains the inductive objects used to instantiate the decision procedure, which is called a *reification*. Without modifying Why3, it is up to the user to provide them. Even for an algorithm as simple to verify as Strassen's, the user might forfeit before finishing to translate all the terms of the algorithm.

Since the function $M \mapsto \ulcorner M \urcorner$ cannot be expressed in the logical language, it is usually defined in a meta-language, either the programming language of the formal system or some dedicated system provided by the system. For example, Coq provides a meta-language named Ltac, in which a reification procedure can be defined (see Section 3.4.1). An approach for Why3 will be described in Section 5.1.2.

Chapter 3

Numerical Quadrature

As a first illustration of the relation between theorems, proofs, programs, and arithmetic, let us consider the case of numerical quadrature. Numerical quadrature represents a family of algorithms for computing an approximation of a proper definite integral

$$\int_u^v f(x)\,dx,$$

of a function $f : \mathbb{R} \to \mathbb{R}$, as well as a bound on the error of this approximation, in the case of reliable computing.

At their core, most of these algorithms work by first sampling the function f at various points of the interval $[u; v]$ and then performing a weighted summation of these values:

$$\int_u^v f(x)\,dx \simeq \sum_{i=0}^{n-1} w_i \cdot f(x_i). \tag{3.1}$$

This leads to numerous algorithms, *e.g.*, Newton-Cotes, Gauss-Legendre, Gauss-Kronrod, Clenshaw-Curtis, which are readily available in mainstream computer algebra systems. From a reliability point of view, Equation (3.1) raises several questions. Indeed, the accuracy of such an algorithm depends on two factors:

1. the inherent discretization due to considering only the points $(x_i)_i$,
2. the way the vector of weights $(w_i)_i$, the vector of values $(f(x_i))_i$, and their scalar product, are computed.

These quadarature methods are designed to be somehow exact on a given family of functions, *e.g.*, polynomials of degree k. Thus, the impact of discretization can be estimated by looking how well f is approximated by polynomials of degree k, which can be obtained by bounding the $k + 1$-th derivative of f on $[u; v]$. Computing such a bound can be a challenge, so most implementations do not go that far and rather rely on heuristics to evaluate their accuracy. They boil down to increasing the number of points $(x_i)_i$ until the value of the approximated integral stops fluctuating. As for the accuracy of the computations themselves, the assumption usually ends up being that *binary64* precision is sufficient for most people.

Unfortunately, some mathematical proofs rely on definite integrals that do not have any known closed formula and thus are evaluated numerically. Some examples are Helfgott's

proof of the ternary Goldbach conjecture [Hel14] or the first proof of the double bubble conjecture [HS00]. What kind of trust can we have in the resulting theorems?

Note that some mathematicians are aware of the shortcomings of numerical computations, as can be seen from a question asked by Helfgott on an Internet forum dedicated to mathematics:[1]

> "I need to evaluate some (one-variable) integrals that neither SAGE nor Mathematica can do symbolically. As far as I can tell, I have two options:
>
> 1. Use GSL (via SAGE), Maxima or Mathematica to do numerical integration. This is really a non-option, since, if I understand correctly, the "error bound" they give is not really a guarantee.
>
> 2. Cobble together my own programs using the trapezoidal rule, Simpson's rule, etc., and get rigorous error bounds using bounds I have for the second (or fourth, or what have you) derivative of the function I am integrating. This is what I have been doing.
>
> Is there a third option? Is there standard software that does (2) for me?"

Chapter 2 has presented some of the ingredients we can use to solve this kind of problem. For effectiveness, we will be using interval arithmetic with floating-point bounds. For additional trust, we will be performing the computations inside the Coq proof assistant. So, we need to formalize floating-point arithmetic and interval arithmetic. But that is not sufficient. We also need to formalize what a definite integral is and to verify an algorithm that computes an enclosure of it. Our goal is to get to the point where we can verify (or disprove) the following inequality taken from Helfgott's proof of the ternary Goldbach conjecture [Hel14, p. 35]. Notice that the integral is actually improper (infinite endpoints), which adds another layer of difficulty.

$$\int_{-\infty}^{+\infty} \frac{(0.5 \cdot \log(\tau^2 + 2.25) + 4.1396 + \log \pi)^2}{0.25 + \tau^2} \, d\tau \leq 226.844. \tag{3.2}$$

Section 3.1 presents the formalization of real analysis available in the standard library of Coq. It also presents the Coquelicot library, a conservative extension of the standard library that is rich enough to express improper definite integrals. Section 3.2 then presents the Flocq library, which can be used to perform floating-point computations inside Coq and to relate the results of these computations to real numbers. This is sufficient to formalize an effective interval arithmetic, which is part of the CoqInterval library and presented in Section 3.3. The trouble with interval arithmetic is the dependency effect, so Section 3.4 details the methods that have been in implemented in CoqInterval to fight it: bisection, automatic differentiation, and Taylor models. Finally, Section 3.5 shows how definite integrals, both proper and improper, are supported by CoqInterval. The formal verification of a bound on the integral of Equation (3.2) will serve as an illustration.

[1] `http://mathoverflow.net/questions/123677/rigorous-numerical-integration`

3.1 Real analysis

The first step is to formalize all the concepts that are needed to express Equation (3.2) as a formal statement in Coq. Neither is Coq the only formal system to come with a formalization of real analysis, nor does it come with the best one, as can be seen from a survey that Sylvie Boldo, Catherine Lelay, and I wrote [BLM16]. A lot of work went into the formalization of both floating-point and interval arithmetics, so this makes Coq the best candidate for our objective. Section 3.1.1 presents the standard library of Coq as well as some of its shortcomings. Section 3.1.2 then presents the Coquelicot library, which extends the formalization of real analysis.

3.1.1 Axiomatization and standard library

Coq's standard library `Reals`[2] axiomatizes real arithmetic, with a classical flavor [May01]. It provides some notions of elementary real analysis, including the definition of continuity, differentiability, and Riemann integrability. It also comes with a formalization of the properties of usual mathematical functions like sin, cos, exp, and so on.

The formalization of real numbers from the standard library is axiomatic rather than definitional. Instead of building real numbers as Cauchy sequences or Dedekind cuts of rational numbers and proving their properties, Coq's developers have assumed the existence of a set with the usual properties of the real line. In other words, the standard library states that there is a set $\mathbb{R}$, some arithmetic operators $-$, $+$, $\times$, $\bullet^{-1}$, and a comparison operator $<$, that have the properties of an ordered field. Except perhaps for the choice of the domain of $\bullet^{-1}$, these axioms are not controversial.

To get the proper real line, one also needs this field to be Archimedean and closed under the supremum bound. A peculiarity of Coq's standard library comes from the fact that these axioms are expressed by two functions: one computes the integer part of a real number, the other computes the supremum of an upper-bounded subset of $\mathbb{R}$. Moreover, there is a third function, which is able to compare two real numbers. Below are the actual definitions from the library:

- Lemma `archimed` states that $\mathsf{up} : \mathbb{R} \to \mathbb{Z}$ satisfies $\forall x \in \mathbb{R},\ x < \mathsf{up}(x) \leq x + 1$.
- Given a subset E of $\mathbb{R}$ and some proofs that it is both inhabited ($\exists x,\ x \in E$) and bounded ($\exists M,\ \forall x,\ x \in E \Rightarrow x \leq M$), function `completeness` returns a real that is the least upper bound of E.
- Given two real numbers x and y, function `total_order_T` tells which of $x < y$, $x = y$, or $x > y$, holds. This is equivalent to the decidability of the equality on real numbers.

In addition to the previous axioms, the standard library sometimes makes use of the excluded middle. The CoqTail project[3], however, has shown that it was unneeded. So, we assume that excluded middle is not available and we will never make use of it, even if, in practice, there are still remnants of it in the standard library.

From these operations and axioms, Coq's standard library then defines limits, derivatives, integrals, and a few elementary functions (exp, log, cos, and so on). The main shortcoming

[2] `https://coq.inria.fr/distrib/current/stdlib/`
[3] `http://coqtail.sourceforge.net/`

with the approach followed by Coq's formalization of real analysis is its extreme reliance on dependent types. Let us consider the example of integrals. The `Reals` library starts by defining a predicate `Riemann_integrable` such that `(Riemann_integrable f u v)` means that $f \in \mathbb{R} \to \mathbb{R}$ is integrable between points u and v. As the name implies, Riemann's criterion is used for integrability, *i.e.*, for all $\varepsilon > 0$, there exist two step functions ψ_1 and ψ_2 such that the difference between f and ψ_1 is bounded by ψ_2 for any point between u and v and such that the integral of ψ_2 does not exceed ε.

While mathematically fine, the Coq definition of `Riemann_integrable` is quite strange in that it lies in `Type` rather than `Prop`, the type of logical propositions. A much more salient issue is that it is then used to define integrals:

```
RiemannInt : forall (f : R -> R) (u v : R), Riemann_integrable f u v -> R.
```

Thus, whenever users would like to talk about $\int_u^v f$, they first have to exhibit a proof that f is integrable between u and v. In practice, this definition makes it especially painful to manipulate. This issue is not specific to integrals and also occurs for differentiability. It would be much more practical if $\int_u^v f$ always existed, whether f is integrable or not, but one could tell something interesting about it only if f is integrable. That principle led to the Coquelicot library.

3.1.2 Coquelicot's approach

Coquelicot[4] was developed by Sylvie Boldo, Catherine Lelay, and me [BLM15, Lel15b]. Among other things, it redefines Riemann's integrability so that it lies in `Prop`. Moreover, it extends it so that it does not accept just functions of $\mathbb{R} \to \mathbb{R}$, but also functions of $\mathbb{R} \to \mathbb{V}$ with $\mathbb{V}$ any normed vector space over $\mathbb{R}$. The predicate `(is_RInt f u v i)` states that f is integrable between points u and v and that its integral is equal to $i \in \mathbb{V}$. The actual definition of integrability will be detailed later. Thanks to this predicate and under the assumption that $\mathbb{V}$ is also complete, we can now define Riemann's integral as follows:

```
Definition RInt {V : CompleteNormedModule R} (f : R -> V) (u v : R) : V :=
  iota (is_RInt f u v).
```

Function `iota` is Hilbert's ι operator, *i.e.*, if there exists one and only one value that satisfies the given predicate, it is returned. Thus, by proving that the integral is unique if it exists, we get the following property:

```
(exists i : V, is_RInt f u v i) -> is_RInt f u v (RInt f u v).
```

Assuming that `is_RInt` actually characterizes integrability, we have thus obtained a *total function* that returns the value of the integral whenever it exists, without having to know *a priori* whether it does. The existence of Hilbert's operator, especially in the constructive setting of Coq, seems quite magical though. The trick is that Coquelicot defines this operator only for a complete space, and that it proves the above property only for a complete normed left-module. Let us see what these notions encompass by giving an overview of Coquelicot's hierarchy.

[4] `http://coquelicot.saclay.inria.fr/`

Coquelicot's hierarchy

A *uniform space* is a set V that provides a notion of *ball* $\mathcal{B}_\varepsilon(x)$. The following definition shows the few properties that balls have to satisfy.

Definition 3.1 (Uniform space)

- $\forall x \in V,\ \forall \varepsilon > 0,\ x \in \mathcal{B}_\varepsilon(x)$;
- $\forall x, y \in V,\ \forall \varepsilon \in \mathbb{R},\ y \in \mathcal{B}_\varepsilon(x) \Rightarrow x \in \mathcal{B}_\varepsilon(y)$;
- $\forall x, y, z \in V,\ \forall \varepsilon_1, \varepsilon_2 \in \mathbb{R},\ y \in \mathcal{B}_{\varepsilon_1}(x) \Rightarrow z \in \mathcal{B}_{\varepsilon_2}(y) \Rightarrow z \in \mathcal{B}_{\varepsilon_1+\varepsilon_2}(x)$.

Once we have a uniform space, we can define what it means for a predicate P to hold in a *neighborhood* of x: $\exists \varepsilon > 0,\ \forall y \in V,\ y \in \mathcal{B}_\varepsilon(x) \Rightarrow P(y)$. If we see P as the set where the property holds, this notion can also be expressed as $\exists \varepsilon > 0,\ \mathcal{B}_\varepsilon(x) \subseteq P$. The corresponding Coquelicot predicate is `(locally x P)`.

The predicate `(locally x)` is actually a *proper filter*, which means that it satisfies the following properties (where $\mathcal{F}$ designates its interpretation as a set of subsets of V):

Definition 3.2 (Proper filter)

- $V \in \mathcal{F}$;
- $\forall P, Q \subseteq V,\ P \in \mathcal{F} \Rightarrow Q \in \mathcal{F} \Rightarrow P \cap Q \in \mathcal{F}$;
- $\forall P, Q \subseteq V,\ P \subseteq Q \Rightarrow P \in \mathcal{F} \Rightarrow Q \in \mathcal{F}$;
- $\emptyset \notin \mathcal{F}$.

In Coquelicot's formalization, the three first properties are packaged in the `Filter` type class. The `ProperFilter` and `ProperFilter'` type classes extend `Filter` so as to include the last property. The difference between these two type classes characterizes whether the last property is constructive, *i.e.*, whether one can exhibit an inhabitant of any set contained in $\mathcal{F}$.

While filters date back to Cartan [Car37b, Car37a], it is a library for the Isabelle/HOL proof assistant that showed us how helpful it could be when formalizing real analysis [HIH13]. Indeed, filters generalize neighborhoods, so we can use them to define a kind of *convergence* by saying that the preimage of any neighborhood is a neighborhood:

Definition 3.3 (Convergence) *Function $h \in T \to U$ is said to converge from filter $\mathcal{F} \subseteq \mathcal{P}(T)$ toward filter $\mathcal{G} \subseteq \mathcal{P}(U)$ if and only if*

$$\forall P \subseteq U,\ P \in \mathcal{G} \Rightarrow h^{-1}(P) \in \mathcal{F}.$$

This relation, denoted `filterlim` in Coquelicot, is at the core of the predicate `(is_RInt f u v i)`. A Riemann integral of f is the limit of the integrals of the step functions $f|_\sigma$ when the pointed subdivision σ becomes arbitrarily fine. We skip over the precise definitions of σ and $f|_\sigma$; let us just say that σ samples $[u; v]$ and that $f|_\sigma$ is the corresponding sampling of f over $[u; v]$. To apply `filterlim`, we instantiate $\mathcal{G}$ using `(locally i)` and h using $\int_u^v f|_\sigma$. As for the filter $\mathcal{F}$, it characterizes the fineness of the pointed subdivision σ, *i.e.*, the maximal distance between two consecutive sampling points. Informally, this could be expressed as follows:

$$\int_u^v f|_\sigma \xrightarrow[|\sigma| \to 0]{} \int_u^v f.$$

An important property of `filterlim` is that, if a function $g : T \to U$ converges from a proper filter $\mathcal{F}$ to two filters `(locally` i_1`)` and `(locally` i_2`)`, then i_1 and i_2 are indistinguishable with respect to the balls of the uniform space V, *i.e.*, $\forall \varepsilon > 0,\ i_2 \in \mathcal{B}_\varepsilon(i_1)$.

We also need some kind of Cauchy criterion to characterize converging functions and talk about their limit. This is provided by the notion of *complete space*. A uniform space V is complete if it provides an operator $\lim : \mathcal{P}(\mathcal{P}(V)) \to V$ that satisfies the following properties:

Definition 3.4 (Complete space)

- $\forall \mathcal{F}$ *proper*, $(\forall \varepsilon > 0,\ \exists x \in V,\ \mathcal{B}_\varepsilon(x) \in \mathcal{F}) \Rightarrow \forall \varepsilon > 0,\ \mathcal{B}_\varepsilon(\lim \mathcal{F}) \in \mathcal{F}$;
- *extensionally equal filters have indistinguishable limits.*

The second property is mostly an artifact due to the lack of extensionability in Coq and thus can be ignored. The first property is more interesting. It states that, in a complete space, if a proper filter contains arbitrarily small balls, then it also contains arbitrarily small balls all centered on the same point, which is defined as the limit of this filter. We can now give an explicit definition of `iota`:

```
Definition iota {V : CompleteSpace} (P : V -> Prop) :=
  lim (fun A => (forall x, P x -> A x)).
```

At this point, we do not have the expected property, but we are getting close. Indeed, if all the points of P are indistinguishable, then `(iota P)` is indistinguishable from any of these points. So, if indistinguishable points were actually equal, we would obtain Hilbert's operator. This will be the case later, once we have a complete normed module.

We have yet to express the weighted sum $\int_u^v f|_\sigma = \sum_{i \leq N} (t_{i+1} - t_i) \cdot f(x_i)$. To do so, we need the set $\mathbb{V}$ to have a structure of $\mathbb{R}$-vector space. Actually, since Coquelicot does not really need to compute the inverse of scalars, a structure of *left-module* (named `ModuleSpace`) is sufficient. An Abelian group $(V, +)$ is a left-module over a ring $(K, +, \times)$, if it provides a product $\cdot : K \times V \to V$ that satisfies the following properties:

Definition 3.5 (Left-Module)

- $\forall u \in V,\ 1 \cdot u = u$;
- $\forall x, y \in K,\ \forall u \in V,\ x \cdot (y \cdot u) = (x \times y) \cdot u$;
- $\forall x \in K,\ \forall u, v \in V,\ x \cdot (u + v) = x \cdot u + x \cdot v$;
- $\forall x, y \in K,\ \forall u \in V,\ (x + y) \cdot u = x \cdot u + y \cdot u$.

A *normed module* V is defined as being both a K-left-module and a uniform space. Moreover, it provides a norm $\| \bullet \| : V \to \mathbb{R}$ and a factor $\theta \in \mathbb{R}$ that satisfy the following properties:

Definition 3.6 (Normed module)

- $\forall x, y \in V,\ \|x + y\| \leq \|x\| + \|y\|$;
- $\forall \ell \in K,\ \forall x \in V,\ \|\ell \cdot x\| \leq |\ell| \cdot \|x\|$;
- $\forall x, y \in V,\ \forall \varepsilon \in \mathbb{R},\ \|y - x\| < \varepsilon \Rightarrow y \in \mathcal{B}_\varepsilon(x)$;
- $\forall x, y \in V,\ \forall \varepsilon > 0,\ y \in \mathcal{B}_\varepsilon(x) \Rightarrow \|y - x\| < \theta \cdot \varepsilon$;

- $\forall x \in V,\ \|x\| = 0 \Rightarrow x = 0$.

The first two properties are the traditional ones. The next two ensure that the norm is compatible with the balls, up to some factor θ. When we combine them with the last property, we now have that indistinguishable points of the uniform space are actually equal. Thus, if the normed module is also complete, we obtain Hilbert's operator and we can use it to define Riemann's integral as a total function.

Fitting $\mathbb{R}$ into the hierarchy

Up to now, we have just manipulated axiomatic structures. Can we actually instantiate all these axioms so that we can interpret $\mathbb{R}$ (as defined in the standard library) as a complete normed module $\mathbb{V}$? Most of the above axioms are trivial to instantiate using Coq's standard library. The difficult part lies in defining $\lim : \mathcal{P}(\mathcal{P}(\mathbb{R})) \to \mathbb{R}$ and proving its first property.

If a filter $\mathcal{F}$ converges toward a point $\ell \in \mathbb{R}$, then ℓ is present in all the subsets of $\mathbb{R}$ that are in $\mathcal{F}$. In particular, it is present in all the open segments $(x; x+2)$ of $\mathcal{F}$, which forces $x \in (\ell - 2; \ell)$. So, we could just define ℓ as the supremum of such values x:

$$\lim \mathcal{F} = \sup\{x \mid \mathcal{B}_1(x+1) \in \mathcal{F}\}.$$

This is where things get a bit awkward. Indeed, the `completeness` axiom requires that the set passed to sup be both upper bounded and nonempty. In a classical setting, the boundedness property is easy: if it was possible to find arbitrarily large values of x such that $\mathcal{B}_1(x+1)$ is in $\mathcal{F}$, taking the intersection of two disjoint balls would contradict the fact that $\mathcal{F}$ is proper. Nonemptyness, however, does not hold. For example, the filter $\{P \mid [0;3] \subseteq P\}$ does not contain any ball of radius one. Anyway, even if the property was holding, we would not be out of the woods yet, since `completeness` requires both properties to hold constructively, *i.e.*, we need to exhibit an element of the set as well as an upper bound. Thus, this approach is a dead end. We need some unrestricted form of `completeness`, *i.e.*, no prerequisite on the set and a result in $\mathbb{R} \cup \{-\infty, +\infty\}$.

It happens that this can be built from the axioms of the `Reals` library. The building block is the *limited principle of omniscience*, or LPO for short. Let P be a decidable predicate on natural numbers. This principle states that one can decide whether the property never holds. Moreover, if $P(n)$ happens to hold for some number n, the LPO produces such a number.[5] The original idea of the proof comes from a formalization by Kaliszyk and O'Connor [KO09]. The CoqTail project later improved it by removing the need for the `not_all_ex_not` corollary of the excluded-middle axiom. I have improved the proof further by getting rid of the sizable amount of analysis it needed (geometric series, logarithm, and so on).

Let us sketch this proof. Thanks to the decidability of P, we build a function $f(n)$ that returns $1/(n+1)$ if $P(n)$ holds and 0 otherwise. Let us consider the subset of real numbers $\{f(n) \mid n \in \mathbb{N}\}$. It is nonempty and bounded by 1, thus its supremum is given by `completeness`. This supremum can be tested against 0 using `total_order_T`. If it is zero, we deduce $\forall n,\ \neg P(n)$. Otherwise we compute its discrete inverse with `archimed`, which is an integer n such that $P(n)$ holds [BLM12].

Now that we can use the LPO, we can define our improved $\sup E$. First, consider the sets $E_n = \{0\} \cup (E \cap (-\infty; n])$. These sets are upper bounded by n and nonempty, so

[5] The LPO does not imply the excluded middle, but it does solve the halting problem.

we can compute the supremum $s_n = \sup E_n$ using `completeness`. If this sequence has an upper bound, then it is also an upper bound of E. Otherwise E is not upper bounded and $\sup E = +\infty$. Applying the LPO to the predicate $n \mapsto (s_n \leq m)$ tells us whether the sequence (s_n) is bounded by m. Let us denote b_m this result. Applying the LPO to $m \mapsto b_m$ either tells us that b_m never holds (in which case (s_n) is not bounded) or it gives us an integer m such that b_m holds, *i.e.*, an upper bound of (s_n).

Let us now assume that m is an upper bound of E. We consider a new sequence of sets: $E'_n = \{-n\} \cup E$. Again, these sets are upper bounded (by m) and nonempty, so we can compute the supremum $s'_n = \sup E'_n$. Note that, if $s'_n \neq -n$ holds for some n, then s'_n is also a supremum of E. Otherwise E is empty, since any $-n$ is an upper bound of it. So it is just a matter of applying the LPO one last time.

We have thus obtained an unrestricted form of sup, without introducing any additional axiom. It is hardly more powerful than the standard one though, since we still cannot prove a property such as $\forall x < \sup E,\ \exists y \geq x,\ y \in E$. Anyway, we do not need this property in order to fit $\mathbb{R}$ into Coquelicot's hierarchy.

Improper integrals

We have shown that, given Coq's standard axioms of real numbers, we can define Riemann's integral as a total function. We now want to extend the definition to the improper case. The bounds are no longer real numbers; they are filters. The advantage of filters is that they can just as well represent finite bounds, infinite bounds, and excluded finite bounds, *e.g.*, 0^+. This generalized integral $\int_{\mathcal{A}}^{\mathcal{B}} f$ is defined as the limit of $\int_a^b f$ when (a, b) converges accordingly to the filter product of $\mathcal{A}$ and $\mathcal{B}$.

As with the proper case, we start by defining a predicate characterizing that the generalized integral exists and is equal to some value, and then we recover this value using operator ι. An early definition of the predicate looked like the following conjunction:

$$\int_{\mathcal{A}}^{\mathcal{B}} f = i \Leftrightarrow \begin{cases} \exists P \in \mathcal{A} \otimes \mathcal{B},\ \forall (a,b) \in P,\ \int_a^b f \text{ exists} \\ \int_a^b \xrightarrow[(a,b)\to\mathcal{A}\otimes\mathcal{B}]{} i. \end{cases}$$

This definition was painful to use in practice, since both conjuncts are more or less the same property. Indeed, the first one talks about the integrability predicate, while the other one talks about the total function, *i.e.*, the composition of the integrability predicate with Hilbert's operator. Moreover, the resulting total function would thus nest two instances of Hilbert's operator. This caused a lot of noise in the proofs.

To circumvent this issue, I have introduced a new notion of convergence. The one based on `filterlim` takes an explicit function of type $T \to U$ while we want a version that takes a function implicitly defined by a relation.

Definition 3.7 (Convergence) *A function $h \in T \to U$ defined by a relation $H \subseteq T \times U$ is said to converge from filter $\mathcal{F} \subseteq \mathcal{P}(T)$ toward filter $\mathcal{G} \subseteq \mathcal{P}(U)$ if and only if*

$$\forall P \subseteq U,\ P \in \mathcal{G} \Rightarrow \{t \in T \mid \exists u \in U,\ (t,u) \in H \wedge u \in P\} \in \mathcal{F}.$$

This definition is named `filterlimi` in Coquelicot and most of the theorems that exist for `filterlim` also exist for `filterlimi`. Obviously, H might represent a partial function (that is the point), but it might also represent a multi-function. Most theorems do not care, though,

so only a few theorems about `filterlimi` have an additional hypothesis to exclude this case. Note also that, as long as $\mathcal{F}$ and $\mathcal{G}$ are proper filters, h cannot converge unless it is defined, so partial functions are not an issue.

This leads to the following definition for the generalized version of Riemann's integral, given a function $f \in \mathbb{R} \to \mathbb{V}$, two filters $\mathcal{A}$ and $\mathcal{B}$, and a value $i \in \mathbb{V}$.

```
Definition is_RInt_gen {V : NormedModule R} (f : R -> V)
                       (A B : (R -> Prop) -> Prop) (i : V) :=
  filterlimi (fun (a, b) => is_RInt f a b) (filter_prod A B) (locally i).
```

To illustrate the use of filters, total functions, and improper integrals, let us consider the equality $\int_{0^+}^{1} x^3 \log^2 x \, dx = 2^{-5}$. It can be stated as follows using Coquelicot:

```
Goal RInt_gen (fun x => x^3 * (ln x)^2)) (at_right 0) (at_point 1) = 1/32.
```

3.1.3 Assessment

Coquelicot does not just come with a formalization of Riemann's integral. It also provides homogeneous definitions and lemmas for limits [Lel15a], derivatives [BLM12], series, and power series [Lel13]. Moreover, the $\mathbb{R}$-instances of these definitions are proved equivalent to those of the standard library, so that user developments can easily mix both libraries [BLM15]. As with integrals, total functions are available for all of those notions of analysis. These functions return the expected value in case of convergence and an arbitrary value otherwise [Lel15b]. Finally, the library comes with a bit of automation regarding the computation of derivatives [LM12].

This brings Coq to the level of other formal systems with respect to real analysis, except for two parts [BLM16]. First, Coquelicot is quite poor when it comes to differential equations. For example, it does not provide any theorem that asserts the existence of a solution to a differential equation. The other missing part is complex analysis. For example, it does not even prove that holomorphic functions are analytic. We could decide that it does not matter for a formalization dedicated to real analysis, but it is not uncommon for results of real analysis to become trivial to prove when considering the complex plane. A common example is the use of Cauchy's integral formula to obtain the value of an improper integral.

3.2 Floating-point arithmetic

As explained in Section 2.2, interval arithmetic is a way to automate verification of properties on real numbers. To implement it, we need some effective arithmetic to perform computations on interval bounds. One such arithmetic is floating-point arithmetic. Since we will be performing operations inside the Coq proof assistant, we do not necessarily need these operations to fully comply with the IEEE-754 standard. We just need them to manipulate integer significands and exponents, and we also need theorems that relate their results to real numbers.

The Flocq formalization,[6] which was developed by Sylvie Boldo and me, can be used for that purpose [BM11, BM17]. This library provides a multi-radix multi-format formalization of floating-point arithmetic. Multi-radix means that the formalization supports binary floating-point numbers, $m \cdot 2^e$, decimal ones, $m \cdot 10^e$, or more exotic ones, $m \cdot 3^e$. Here, we will be

[6] `http://flocq.gforge.inria.fr/`

using radix 2 for efficiency reasons, but none of the theorems actually depend on it. The only constraint, sometimes, will be that the radix needs to be even, so that an interval midpoint can be computed exactly.

Multi-format means that Flocq encompasses various formats into a single formalism: fixed-point arithmetic, floating-point arithmetic with unbounded exponents, floating-point arithmetic with gradual underflow, floating-point arithmetic with abrupt underflow, and so on. Again, since we intend to perform the computations inside the logic of the formal system, we do not have any hardware constraint. So, we go for an arithmetic with unbounded exponents. The precision of the computations, however, will be set on a per-operation basis.

Among the many components of Flocq, two of them are of interest to us, here.[7] The first one is responsible for defining formats (Section 3.2.1) and rounding operations on real numbers (Section 3.2.2). The second one, described in Section 3.2.3, gives a computational meaning to these operations when applied to floating-point numbers (seen as pairs of integers).

3.2.1 Formats

The very first component is the definition of a radix. It is just an integer that is larger than or equal to two, to avoid some degenerate cases:

```
Record radix : Set := Build_radix
  { radix_val : Z;
    radix_prop : Z.leb 2 radix_val = true }.
```

There are two important functions that are related to radices. The `bpow` function takes a radix β and an integer exponent e and returns β^e. The `mag` function performs the converse operation: given a radix and a real number x, it returns the unique integer e such that $\beta^{e-1} \leq |x| < \beta^e$ holds. Irrespective of a format, a floating-point number is just a pair `float` of two integers:

```
Record float (beta : radix) : Set := Float
  { Fnum : Z;  Fexp : Z }.
```

This pair (m, e) is given its interpretation as a real number $m \cdot \beta^e$ by the `F2R` function:

```
Definition F2R (beta : radix) (f : float beta) : R :=
  (IZR (Fnum f) * bpow beta (Fexp f))%R.
```

Flocq generalizes all the formats into a single framework through the use of exponent functions: $\varphi \in \mathbb{Z} \to \mathbb{Z}$. Given the exponent of a real number (as returned by `mag`), they compute the exponent of the unit in the last place (ulp) for this number to fit in the format. For example, for a floating-point format with unbounded exponents and with precision ϱ, the function will be as simple as $\varphi(e) = e - \varrho$ (denoted `FLX_exp` below).

The `cexp` function takes a radix, an exponent function, and a real number x, and it returns the exponent of the ulp of x, assuming it is in the format. This is the *canonical exponent* of x:

```
Definition cexp (beta : radix) (fexp : Z -> Z) (x : R) : Z :=
  fexp (mag beta x).
```

Similarly, the `scaled_mantissa` function returns the corresponding significand of x, which would be an integer if x was in the format.

[7] Some other parts of Flocq will be presented in Sections 4.3.2 and 5.2.

```
Definition scaled_mantissa (beta : radix) (fexp : Z -> Z) (x : R) : R :=
  (x * bpow beta (- cexp beta fexp x))%R.
```

By using the `Ztrunc` function which truncates a real number into an integer, we then get the definition of a format in Flocq. A number x is part of the format if it is left unchanged by truncating its significand at the position mandated by the exponent function representing the format:

```
Definition generic_format (beta : radix) (fexp : Z -> Z) (x : R) : Prop :=
  x = F2R (Float beta
    (Ztrunc (scaled_mantissa beta fexp x))
    (cexp beta fexp x)).
```

While fully generic, this definition is a bit cumbersome to use. So, when a format is fixed, it is better to use some dedicated predicate. For example, Flocq provides an `FLX_format` predicate to represent floating-point formats with unbounded exponents and a fixed precision ϱ. A number is part of the format if it can be represented as $m \cdot \beta^e$ with $|m| < \beta^\varrho$:

```
Inductive FLX_format (beta : radix) (prec : Z) (x : R) : Prop :=
  FLX_spec : forall f : float beta,
    x = F2R f -> (Z.abs (Fnum f) < beta ^ prec)%Z -> FLX_format beta prec x.
```

Flocq then provides theorems to switch from one format representation to the other, such as the following one:

```
Theorem generic_format_FLX :
  forall (beta : radix) (prec : Z) (x : R),
  FLX_format beta prec x ->
  generic_format beta (FLX_exp prec) x.
```

3.2.2 Rounding operators

Now that we can characterize real numbers that are part of a format, let us see how to round those who are not. Since representable numbers have a significand that is an integer, rounding a non-representable number should be as simple as rounding its significand to an integer. That is the purpose of the `round` function. In addition of β, φ, and x, it also takes an argument *rnd* that tells how to round to an integer:

```
Definition round (beta : radix) (fexp : Z -> Z) (rnd : R -> Z) (x : R) : R :=
  F2R (Float beta
    (rnd (scaled_mantissa beta fexp x))
    (cexp beta fexp x)).
```

Choosing `Zfloor` for *rnd* will round toward $-\infty$ and choosing `Zceil` will round toward $+\infty$. These are the two rounding directions that will be needed for interval arithmetic.

As with `generic_format`, the definition of `round` is not immediately applicable and it is better to have specific relations for the more common rounding directions. For example, the following one characterizes rounding toward $-\infty$. Given a format (seen as a subset of $\mathbb{R}$), two real numbers x and f are related if f is the maximum of all the floating-point numbers of the format smaller than or equal to x:

```
Definition Rnd_DN_pt (F : R -> Prop) (x f : R) : Prop :=
  F f /\ (f <= x)%R /\
  forall g : R, F g -> (g <= x)%R -> (g <= f)%R.
```

Flocq then provides some theorems that link the generic operator `round` with some specific relations such as `Rnd_DN_pt`:

```
Theorem round_DN_pt :
  forall (beta : radix) (fexp : Z -> Z), Valid_exp fexp ->
  forall x : R,
  Rnd_DN_pt (generic_format beta fexp) x (round beta fexp Zfloor x).
```

For interval arithmetic, we only need $\bigtriangledown(x)$ to be less than or equal to x. We do not need the two other properties provided by `Rnd_DN_pt`: $\bigtriangledown(x)$ is in the format and it is the largest such one.[8] Still, these properties are important for other use cases, *e.g.*, when formalizing IEEE-754 arithmetic.

Unfortunately, there is *a priori* no reason for a rounded value $\square(x)$ to be in the format described by φ. Indeed, by definition of `round`, $\square(x)$ is a multiple of $\beta^{\mathsf{cexp}(x)}$. But to be part of the format, it needs to be a multiple of $\beta^{\mathsf{cexp}(\square(x))}$. Most of the time, the canonical exponents of x and $\square(x)$ are equal, so $\square(x)$ is trivially part of the format. One problematic case, however, is $\beta^{e-1} \leq |x| < \beta^e = |\square(x)|$. Indeed, if $\varphi(e+1) > e$, then $\square(x)$ will not be part of the format. Another problematic case is when x lies in the range of subnormal numbers. For both cases, we can find examples of φ functions that cause `round` to not behave properly. The purpose of the `Valid_exp` predicate in the statement of `round_DN_pt` is to exclude these degenerate formats. That restriction is not a severe one, though, as even floating-point formats with abrupt underflow are accepted by `Valid_exp`.

3.2.3 Effective operations

Up to now, we have only manipulated real numbers, so we cannot perform any computation. Yet, it seems that the definition of `round` could easily be made computable by restricting it to floating-point numbers, representable or not. For example, given two floating-point numbers, their infinitely precise sum and product are also floating-point numbers, which could then be rounded to fit in a target format.

For division and square root, the situation is slightly more complicated, since an infinitely precise result might not always be expressible as $m \cdot \beta^e$. That is why Flocq introduces the notion of location. A real number x lies between two other numbers d and u with a location ℓ if it satisfies the predicate `(inbetween d u x l)`, defined as follows.

```
Inductive inbetween (d u x : R) : location -> Prop :=
  | inbetween_Exact : x = d -> inbetween d u x loc_Exact
  | inbetween_Inexact : forall l : comparison,
    (d < x < u)%R -> Rcompare x ((d + u) / 2) = l ->
    inbetween d u x (loc_Inexact l).
```

The location tells whether x is exactly d, or where it is located with respect to the midpoint between d and u. For floating-point numbers, a specialization of this relation is provided. It specifies how x is located with respect to $m \cdot \beta^e$ and $(m+1) \cdot \beta^e$:

[8] In fact, the latter guarantees the accuracy of some interval operations, so it is still useful from an informal point of view.

```
Definition inbetween_float (beta : radix) (m e : Z) (x : R) (l : location) : Prop :=
  inbetween (F2R (Float beta m e)) (F2R (Float beta (m + 1) e)) x l.
```

Assuming that e is no larger than the canonical exponent of x, knowing m, e, and ℓ is sufficient to decide how x should be rounded, for the standard rounding directions. More details can be found from the book [BM17, Ch. 3]. When rounding toward $\pm\infty$, we do not even need to know where x is located with respect to the midpoint, but it is important when rounding to nearest.

Thus, the matter is now to compute m, e, and ℓ, when performing a division or a square root. Let us consider the case of the division. Flocq provides the helper function `Fdiv_core` for that purpose. Given two numbers $m_1 \cdot \beta^{e_1}$ and $m_2 \cdot \beta^{e_2}$ and a target exponent e, the function returns a pair (m, ℓ) such that `(inbetween_float m e r l)` holds when r is the exact quotient. This is the stated by the following theorem:

```
Theorem Fdiv_core_correct :
  forall (beta : radix) (m1 e1 m2 e2 e : Z),
  (0 < m1)%Z -> (0 < m2)%Z ->
  let (m, l) := Fdiv_core beta m1 e1 m2 e2 e in
  inbetween_float beta m e (F2R (Float beta m1 e1) / F2R (Float beta m2 e2)) l.
```

The function is defined as follows. It first shifts one of the integer significands so that $(m_1 \cdot \beta^{e_1})/(m_2 \cdot \beta^{e_2}) = (m_1'/m_2') \cdot \beta^e$. It then performs a Euclidean division of m_1' and m_2'. The quotient m is part of the returned result. Comparing the remainder r to $\beta/2$ tells where the exact result is located with respect to the midpoint of $m \cdot \beta^e$ and $(m+1) \cdot \beta^e$:

```
Definition Fdiv_core (beta : radix) (m1 e1 m2 e2 e : Z) : Z * location :=
  let (m1', m2') :=
    if (e <=? e1 - e2)%Z
    then ((m1 * beta ^ (e1 - e2 - e))%Z, m2)
    else (m1, (m2 * beta ^ (e - (e1 - e2)))%Z) in
  let (m, r) := Z.div_eucl m1' m2' in
  (m, new_location m2' r loc_Exact).
```

All that is left is to compute e before calling the function. Computing the canonical exponent of the infinitely precise quotient might be complicated, but we have some leeway since e can be smaller. In fact, we can even choose e small enough so that only the first branch of `Fidv_core`'s conditional is ever needed. For example, in the case of an FLX format where m_1 and m_2 both have ϱ digits, choosing $e = e_1 - e_2 - \varrho$ satisfies all the conditions.

The division that has been presented is kind of an ideal operator, since the exponent of the result can never overflow. But Flocq also provides operators that support infinities and NaNs, so as to more closely match the IEEE-754 standard. They will be described in Section 4.3.2.

3.2.4 Assessment

While division and square root have rather complicated algorithms, that is not the case of addition and multiplication. In order to compute $\square(u \cdot v)$ with $u = m_u \cdot \beta^{e_u}$ and $v = m_v \cdot \beta^{e_v}$, Flocq's approach is to first compute $(m_u \cdot m_v) \cdot \beta^{e_u+e_v}$ and then to round it accordingly to the format and direction. Let us denote d_u and d_v the number of β-digits of m_u and m_v respectively. If we assume that the rounded significand is ϱ digits long and that the multiplication is naive, then the asymptotic time complexity is $O(d_u \cdot d_v + \varrho)$. So, if all

the intermediate results are represented using an FLX format with ϱ digits, the complexity is $O(\varrho^2)$. Thus, despite its simplicity, the multiplication algorithm is asymptotically well-behaved, as its complexity depends only on the working precision ϱ. Flocq's addition, however, does not behave that well. Indeed, as explained in Section 2.1.3, it shifts both significands until they are aligned with respect to the smallest exponent. This means that the asymptotic complexity is now $O(\max(d_u + e_u, d_v + e_v) - \min(e_u, e_v) + \varrho)$. Assuming an FLX format, this simplifies to $O(|e_u - e_v| + \varrho)$. This time, the complexity depends not only on the precision, but also on the difference of exponents. That is because Flocq's implementation is too naive. Neither floating-point units nor a multi-precision library like MPFR suffer from this issue.

Another shortcoming of Flocq is that, while the formalization is multi-radix, it provides hardly any algorithm for conversion between formats with differing radices, *e.g.*, from decimal to binary. This matters because CoqInterval internally uses a binary format for efficiency, yet most users expect inputs and outputs to be available as decimal numbers. A decimal number $x = m \cdot 10^e$ can easily be converted to binary by performing a rounded multiplication $\square((m \cdot 2^e) \cdot 5^e)$ if $e \geq 0$, and a rounded division otherwise. So, the larger $|e|$ is, the longer it takes to convert a number. An algorithm that instead performs $\square((m \cdot 2^e) \cdot \square(5^e))$ would be much better behaved from a complexity point of view, but it would be incorrect. Fortunately, it can be made correct [Cli90]. Note that converting from decimal to binary also matters when writing a compiler (see Section 4.3).

3.3 Interval arithmetic

Once we have implemented floating-point operations with directed rounding, the first interval operations are easy to derive and to verify. Section 3.3.1 gives a few implementation details about these basic arithmetic operators. Section 3.3.2 then describes how enclosures of elementary functions are computed. Finally, Section 3.3.3 tells a few words on some low-level efficiency aspects.

3.3.1 Arithmetic operators

As explained in Section 2.2.2, the implementation and verification of interval addition and subtraction is straightforward. The way CoqInterval represents infinite bounds is a bit peculiar, though [Mel08b]. To do so, the library extends the floating-point type with a special element $\bot_{\mathbb{F}}$. When used as a lower bound, this element represents $-\infty$, and as an upper bound, it represents $+\infty$. The creation and propagation rules are that of NaN, *e.g.*, $\square(0 \cdot \square(1/0)) = \bot_{\mathbb{F}}$.

For interval multiplication, we stray away a bit from the description of Section 2.2.2. Indeed, Coq does not offer any way to perform parallel computations, so CoqInterval's implementation does not compute all eight directed products of input bounds before using their minimum and maximum as output bounds. Instead, it studies the sign of the input intervals so that, in most cases, only two products of bounds are needed for computing an interval product. Below are two representative cases for the product between two intervals $[\underline{u}, \overline{u}]$ and $[\underline{v}, \overline{v}]$:

$$\begin{array}{ll} [\bigtriangledown(\overline{u}\underline{v}); \bigtriangleup(\underline{u}\underline{v})] & \text{when } \overline{v} \leq 0, \\ [\min(\bigtriangledown(\underline{u}\overline{v}), \bigtriangledown(\overline{u}\underline{v})); \max(\bigtriangleup(\underline{u}\underline{v}), \bigtriangleup(\overline{u}\overline{v}))] & \text{when } \underline{u} < 0 < \overline{u} \text{ and } \underline{v} < 0 < \overline{v}. \end{array}$$

As in most interval libraries based on floating-point arithmetic, interval multiplication needs to handle infinite bounds with care. Indeed, the tightest result for the interval product

$(-\infty; +\infty) \cdot [0; 0]$ is $[0; 0]$, but performing the floating-point product $\square(0 \cdot \bot_{\mathbb{F}}) = \bot_{\mathbb{F}}$ would lead to infinite and thus grossly overestimated bounds.

Division is implemented using a similar sign-based approach as multiplication, but there is an additional difficulty. Indeed, division, as well as square root, are partial functions, from a mathematical point of view. Thus, it is important to keep track of input values that are out of function domains (see Section 3.5.1 for an illustration). In the case of CoqInterval, a special element $\bot_{\mathbb{I}}$ is returned when input intervals are partially or totally out of the domain. This element is then propagated along subsequent computations. Thus, it plays a role similar to NaN in floating-point arithmetic. To illustrate these concepts, let us have a look at the CoqInterval implementation of interval subtraction.

```
Definition sub prec xi yi :=
  match xi, yi with
  | Ibnd xl xu, Ibnd yl yu =>
    Ibnd (F.sub rnd_DN prec xl yu) (F.sub rnd_UP prec xu yl)
  | _, _ => Inan
  end.
```

Function `F.sub` denotes the floating-point subtraction. It takes two floating-point arguments, as well as a rounding mode and a precision, and returns a correctly rounded floating-point result. The rounding modes `rnd_DN` and `rnd_UP` represent $\bigtriangledown$ and $\bigtriangleup$, respectively. The interval subtraction takes two interval arguments, as well as a precision, and it returns an interval. Intervals are an inductive datatype with two constructors: `Ibnd` represents a closed connected set of real numbers by two floating-point bounds, while `Inan` represents $\bot_{\mathbb{I}}$.

The corresponding correctness theorem is `sub_correct`. It states that the above definition satisfies the containment property. The proof is straightforward, as it is mainly a matter of handling all the $\bot_{\mathbb{I}}$ and $\bot_{\mathbb{F}}$ cases until we are down to plain real arithmetic.

3.3.2 Elementary functions

When it comes to elementary functions, the situation is slightly more complicated. We no longer have relatively simple floating-point operators upon which we can build. Thus, the first step is to implement and verify such operators.

Floating-point operators

Let us make it clear what the signature of these operators is, as we will not be able to achieve correct rounding, except for trivial inputs, *e.g.*, $\exp 0 = 1$. Indeed, the traditional strategy for implementing a mathematical function is, for a given input, to compute more and more precise approximations of the ideal result until we are sure that only one single floating-point number is closest to it (for some precision and rounding mode) [Ziv91]. Unfortunately, this would require us to not only formally verify that CoqInterval's algorithms can actually compute arbitrarily accurate approximations of mathematical functions, but also to design and formally verify an algorithm that computes an upper bound on the number of iterations needed to reach correct rounding. Thus, CoqInterval only provides algorithms that compute enclosures of the mathematical results, and nothing will be guaranteed about the width of these enclosures. Nonetheless, the algorithms will try to return intervals whose relative width matches the user-specified precision. This choice considerably reduces the proof effort. To

summarize, CoqInterval's floating-point elementary functions take a precision and a floating-point input, and return an interval: $\mathbb{N} \to \mathbb{F} \to \mathbb{I}$.

These functions are evaluated using simple power series whose terms happen to be alternating and decreasing for small inputs, *e.g.*, $|x| \leq \frac{1}{2}$. Indeed, we have the following inequalities for such series:

$$0 \leq (-1)^n \left(f(x) - \sum_{k=0}^{n-1} (-1)^k a_k x^k \right) \leq a_n x^n.$$

For example, arctan is given by the following slowly converging power series:

$$\arctan x = x \cdot \sum_{i=0}^{\infty} (-1)^i \cdot \frac{(x^2)^i}{2i+1}.$$

From $|x| \leq \frac{1}{2}$, we can compute how many terms are needed for the remainder of the series to be smaller than a threshold $\beta^{-\varrho}$. For example, when $\beta \leq 4$, k terms are sufficient. This does not mean that the algorithm will compute that many terms. It just means that k is the decreasing argument needed to define the recursive summation. To decide when to stop, the algorithm tests the current remainder against the threshold. Since there is a factor x in front of the series, the absolute bound on the remainder becomes a relative error on the final value [Mel12].

The evaluation of the truncated power series is performed using interval arithmetic, so as to avoid the need for a careful error analysis. Note that the dependency effect is at its worst here, due to the alternated signs during the summation (see Section 2.2.3). Thus, the result would be meaningless if we were to perform the computations with a large interval enclosing x. That is why we are using this approach to approximate the mathematical function at a single point only. That way, all the intervals that appear during the computations are *a priori* minuscule, as their growth is only caused by rounding errors, so the dependency effect hardly matters.

Argument reduction

To get the power series to converge, we need its argument to be close enough to zero, and we need it to be even closer to reduce the amount of terms needed for the remainder to be small enough. This means we need to devise an *argument reduction*.

In the case of $\arctan x$, the argument x is first reduced to $x > 0$ by using the parity of the function. It is then further reduced to $[-\frac{1}{3}; \frac{1}{2}]$ using the following formulas:

$$\arctan x = \begin{cases} \frac{\pi}{4} + \arctan \frac{x-1}{x+1} & \text{for } x \in [\frac{1}{2}; 2], \\ \frac{\pi}{2} - \arctan \frac{1}{x} & \text{for } x \geq 2. \end{cases}$$

Notice that reconstructing the result of arctan involves the constant $\frac{\pi}{4}$ (or $\frac{\pi}{2}$). It is computed thanks to Machin's formula, $\frac{\pi}{4} = 4 \cdot \arctan \frac{1}{5} - \arctan \frac{1}{239}$, which does not require any argument reduction when evaluating arctan. Note that this formula is efficient only when enclosing π at low precision, since the power series of arctan does not converge that fast at point $\frac{1}{5}$. For larger precision, more intricate approaches would have to be formalized [BRT18].

For exp and log, the first reduction step makes sure that the series will later be alternated:

$$\begin{array}{rcll} \exp x & = & 1/(\exp(-x)) & \text{for } x > 0, \\ \log x & = & -\log(x^{-1}) & \text{for } x < 1. \end{array}$$

Notice that, while the argument reduction for exp is error-free when performed using floating-point arithmetic, the one for log is not. It is especially sensitive for values of x close to 1^-. Indeed, the final result is then about

$$\log(1-h) \simeq -\log((1-h)^{-1}(1+\varepsilon)) \simeq h - \varepsilon,$$

with $|\varepsilon| \leq \beta^{-\varrho'}$ the relative error induced by computing the inverse of $1-h$ at some precision ϱ'. Thus, to reach a final accuracy of about ϱ digits, the intermediate precision ϱ' has to be at least $\varrho - \log_\beta h$. This means that CoqInterval's implementation of log is quite inefficient for inputs close to 1^-, especially when the input format is much more precise than the output format.

After the first parity-based reduction, a multiplicative reduction is performed to bring the argument close to 0 for exp and close to 1 for log. This reduction eases the proof process but it is less efficient than the additive reduction implemented in most mathematical libraries [Mul16]. It relies on the following identities:

$$\begin{aligned} \exp x &= (\exp(x/2))^2, \\ \log x &= 2\log\sqrt{x}. \end{aligned}$$

The objective is to bring x into $[-2^{-8}; 0]$ for exp, and into $[1; 1+2^{-8}]$ for log, so as to hasten the power series convergence. These bounds were obtained experimentally [Mel12].

For cos and sin, power series are used when the input is in $[-\frac{1}{2}; \frac{1}{2}]$. For tan, the following identity is used for $x \in [-\frac{1}{2}; \frac{1}{2}]$:

$$\tan x = \frac{\sin x}{\sqrt{1-\sin^2 x}}.$$

Outside $[-\frac{1}{2}; \frac{1}{2}]$, angle-halving formulas are used to get a reduced input:

$$\begin{aligned} \cos x &= 2\cdot(\cos\tfrac{x}{2})^2 - 1, \\ \operatorname{sign}(\sin x) &= \operatorname{sign}(\sin\tfrac{x}{2})\cdot\operatorname{sign}(\cos\tfrac{x}{2}). \end{aligned}$$

These formulas give $\cos x$ and the sign of $\sin x$ for any x. The results of sin and tan are then reconstructed thanks to the following formulas:

$$\begin{aligned} \sin x &= \operatorname{sign}(\sin x)\cdot\sqrt{1-(\cos x)^2}, \\ \tan x &= \operatorname{sign}(\sin x)\cdot\operatorname{sign}(\cos x)\cdot\sqrt{(\cos x)^{-2}-1}. \end{aligned}$$

To compensate for the loss of accuracy due to the reconstruction, the precision of the intermediate computations is increased by about $\log|x|$ digits.

Interval operators

As mentioned earlier, we cannot make use of alternated power series to compute enclosures over non-point intervals. While correct, the results would be meaningless, due to the dependency effect. Thus, as with basic arithmetic operators, we have to make use of monotony properties when devising the interval operators. For exp, log, arctan, power, and integer part, this is straightforward.

For cos and sin, the situation is slightly more complicated. Since we are not using an additive argument reduction, we do not know how the function varies between the bounds

of the input interval. So, CoqInterval has a pragmatic approach. First, sin is monotone on $[-\frac{\pi}{2}; \frac{\pi}{2}]$, so the interval extension is straightforward there. If the input interval is (partly) outside $[-\frac{\pi}{2}; \frac{\pi}{2}]$, then we implement the interval extension of sin using the one of cos, by translating the input interval by $\pm\frac{\pi}{2}$.

Function cos is even and it is monotone on $[0; \pi]$ and on $[\pi; 2\pi]$. So, the implementation of a tight interval extension is mostly straightforward on $[-2\pi; 2\pi]$. If the input interval is outside this domain, CoqInterval first checks whether its width is larger than 2π. In that case, the result is $[-1; 1]$. Otherwise, it uses the fact that cos is a Lipschitz function as follows:

$$\cos[\underline{u}; \overline{u}] = \left(\cos\frac{\underline{u} + \overline{u}}{2} + \frac{\overline{u} - \underline{u}}{2} \cdot [-1; 1]\right) \cap [-1; 1].$$

This is a terribly poor interval extension of cos, but it is isotone. As a consequence, it will not cause any trouble for the methods described in Section 3.4, since bisection can always be applied. Moreover, except for their error interval, Taylor models only need to evaluate functions at point intervals, and thus are not impacted.

3.3.3 Architecture

CoqInterval provides several abstraction layers so that the user can replace some component by another. For example, the interval arithmetic component does not depend on a specific implementation in order to represent bounds. Any implementation will do, as long as it provides the requested floating-point operations. CoqInterval comes with two such implementations. One is a straightforward but slow implementation inherited from Flocq (see Section 3.2.3). The other one uses abstract types to represent the integer significand and exponent of floating-point numbers. Indeed, for a given floating-point radix, it might be worth using some special encoding of integer significands that provides fast shift operations.

CoqInterval offers two components that implement these abstract types in the case of radix 2. One just uses the encoding of positive integers by little-endian lists of bits, *i.e.*, the `Z` type of Coq's standard library. Obviously, lists of bits make the implementation of shifts especially fast. The other component represents integer as binary trees whose leaves are *limbs*, *i.e.*, fixed-width integer [GT06]. These limbs are 31-bit integers, and instead of performing bit-by-bit operations on them, the VM engine of Coq forwards them to the processor so that hardware arithmetic units can be used [AGST10]. This provides very fast arithmetic operations, since multiplication, division, and square root, are sub-quadratic algorithms thanks to the use of binary trees, and the constant factor of their asymptotic complexity is quite low due to the use of the processor. The `interval` tactic uses this arithmetic, unless the user asks for different components.

Memoization

CoqInterval makes use of the VM engine not only to speed integer computations, but also to perform some *memoization*. Indeed, we do not want to have to recompute π whenever we reconstruct some result of the arctan function. So, we need some way to store the previously computed results of the $\mathbb{Z} \to \mathbb{I}$ function that approximates it. This is done using *coinductive* types. Indeed, since a coinductive object is potentially infinite, it cannot be eagerly normalized. So, Coq represents such an object as the value returned by a nullary function. Moreover, once a nullary function has been evaluated, its result is stored, so that Coq does not

have to evaluate it again the next time it accesses the value of the coinductive object, which is precisely memoization. To benefit from this mechanism, CoqInterval stores constants as coinductive infinite lists of coinductive cells, and the k-th cell stores the constant at precision $31k$. Thus, whenever a constant at precision ϱ is needed, Coq will unfold the infinite list up to its $\lfloor \varrho/31 \rfloor$-th cell,[9] and then evaluate the corresponding nullary function, if it has not done so previously [Mel08a]. For example, querying π at precision 42 or 53 both returns an enclosure of π computed with a precision of 62 bits.

3.3.4 Assessment

Special values and decorations

Section 3.3.1 explained that, whenever an interval operator is applied to enclosures that are not contained in the domain of a function, CoqInterval returns $\bot_{\mathbb{I}}$. This behavior is a bit too coarse, as computations could have continued by considering only the values of the input intervals that are part of the function domain. Indeed, the user might have a proof that all the computations are properly defined. Thus, if some inputs are out of the function domains, because of directed rounding or the dependency effect, they can be ignored. So, it depends on whether the interval computations are meant to be a proof of well-definedness or not. The IEEE-1788 standard solves this issue by adding *decorations* to intervals, so that both use cases are covered at once [IEE15]. These decorations degrade along computations, as interval operators get applied to inputs outside continuity or definition domains. In the case of CoqInterval, the meaningful decorations are *trivial* (`trv`) to replace $\bot_{\mathbb{I}}$, and *defined* (`def`) otherwise.

Hardware support

Another specificity of the IEEE-1788 standard is that it provides intervals with hardware floating-point bounds. This is not the case of CoqInterval, which relies on a multi-precision floating-point arithmetic. More precisely, as explained in Section 3.3.3, the default implementation of the tactic relies on a floating-point arithmetic built on top of a fast integer arithmetic, for the sake of performance. As a consequence, the user has to trust not only Coq's kernel, but also the integer arithmetic unit of the processor. That is not too much of a leap of faith, since a faulty processor would presumably prevent Coq from running reliably anyway. What if the user is also willing to trust the floating-point arithmetic unit of the processor? Then it would make sense for CoqInterval to provide a component that provides a fixed-precision floating-point arithmetic, *e.g.*, *binary64*. This would require the following modifications to Coq. First, the VM engine would have to provide opcodes for performing floating-point operations. Second, a Coq library would have to give some useful semantics to these operations. Flocq could be used for that (see Section 4.3.2). These changes could make interval arithmetic in the setting of Coq one or two orders of magnitude faster. This would still be much slower than native tools, but the cost of using formally verified algorithms would become a lot less distressing.

[9] Using a coinductive tree would have permitted to access the cell in logarithmic rather than linear time.

Elementary functions

The implementation of elementary functions is another place of CoqInterval that would benefit from some more work. Using a multiplicative argument reduction is quite naive, but it makes sense for multi-precision arithmetic in a context where proofs are costly. Still, CoqInterval's implementation is especially naive. For example, using repeated square roots to bring an input close to 1 requires a large precision as the information progressively moves toward the least-significant bits. It would be much more efficient, precision-wise, to turn $\sqrt{1+x}-1$ into a floating-point operator. This is the same kind of rationale that leads mathematical libraries to propose not only exp and log but also $\exp x - 1$ and $\log(1+x)$ [IEE08, §9.2]. As for an additive argument reduction, it would become really useful once CoqInterval supports intervals with hardware floating-point bounds. Indeed, once the range and the precision of the inputs are known, memoizing the reduction constants becomes sensible.

Argument reduction is not the only shortcoming of the implementation of elementary functions, the power series evaluation is quite naive too. First, it uses interval arithmetic rather than just floating-point arithmetic. This makes the proofs simpler, as it avoids performing an error analysis, but it means twice as many floating-point operations are used. In the case of a known precision, *e.g.*, hardware floating-point numbers, it would make sense to use polynomials that approximate the function much more uniformly on the reduced domain. While Remez approximants are presumably too costly to compute on the fly, even with memoization, it might make sense to use Chebyshev approximants. In the case of multi-precision, rather than using floating-point or interval arithmetic to evaluate the power series, it might be worth going for *binary splitting*, which means that only exact integer computations are performed to compute the dividend and divisor of the truncated sum [Kar91]. Moreover, the terms of the power series are summed in a way that ensures that only similarly sized integers are multiplied together. Another advantage is that no error analysis is needed since the only rounding error occurs when dividing the dividend by the divisor to recover a floating-point or interval result.

Finally, from a user perspective, the most important shortcoming of CoqInterval is the lack of mathematical functions. This is mostly due to the cost of supporting new functions. The process currently requires to find an argument reduction to a domain where the function is represented by an alternated power series and then to implement the evaluation of this series using interval arithmetic. So, this is quite restrictive, since such an argument reduction might not exist, or it might not be implementable using already implemented functions. Yet, despite all these restrictions, correctness proofs were still cumbersome to perform. It would be worth switching to a more general framework, *e.g.*, representing mathematical functions as solutions of linear differential equations. Then, various approaches exist to produce power series, find truncation orders, perform analytic computations, and so on [Mez10]. Moreover, it would make it possible to tackle not just elementary functions, but also *special* functions, which have much less useful identities available for argument reduction, if any.

3.4 Automated proofs

As explained in Section 3.3.1, CoqInterval provides some interval operators, as well as the theorems stating that they satisfy the containment property. Before we automate the process, let us describe what a manual proof would look like. Suppose that we want to prove $1+\sqrt{x} \leq 4$ when $3 \leq x \leq 5$. We need to apply the containment theorems until the goal has been turned

```
Module F := SpecificFloat StdZRadix2.
Module I := FloatIntervalFull F.
Module J := IntervalExt I.
Notation "x \in xi" :=
  (contains (I.convert xi) (Xreal x)) (at level 100).

Lemma foo x (H : 3 <= x <= 5) : 1 + sqrt x <= 4.
Proof.
refine (proj2 (subset_contains (I.convert _)
    (I.convert (I.bnd F.nan (F.fromZ 4)))
    (I.subset_correct _ _ _) (Xreal _) _)) ; cycle 1.
(* 1 + sqrt x \in ?xi  *) apply (J.add_correct 10%Z).
(* 1 \in ?xi           *) apply I.fromZ_correct.
(* sqrt x \in ?yi      *) apply (J.sqrt_correct 10%Z).
(* x \in ?xi *) apply (H : x \in I.bnd (F.fromZ 3) (F.fromZ 5)).
(* I.subset ... = true *) vm_compute. apply eq_refl.
Qed.
```

Figure 3.1 – Manual application of interval theorems.

into a single interval computation, which can then be run. This leads to the Coq script shown in Figure 3.1.

The script starts by instantiating a few modules: `F` for radix-2 floating-point operations, `I` for interval operations, `J` for friendlier containment facts. Intermediate goals are displayed as comments before the tactics applied to them. To make them more readable, the notation "`x \in xi`" is used to represent an enclosure $x \in \mathbf{x}$. The first tactic turns the goal into an interval problem, *i.e.*, $1 + \sqrt{x} \leq 4$ holds if one can exhibit $\mathbf{y}$ such that both $1 + \sqrt{x} \in \mathbf{y}$ and $\mathbf{y} \subseteq (-\infty; 4]$ hold. Note that, since we have not yet told Coq what $\mathbf{y}$ is, an existential variable `?xi` represents the hole. The next tactics progressively fill this existential variable. For example, applying the containment property for addition, *i.e.*, `(J.add_correct 10%Z)`, tells Coq that $\mathbf{y}$ is obtained by summing the intervals enclosing 1 and $\sqrt{x}$ with a precision of 10 bits. Similarly, the next two tactics tell how to handle 1 and $\sqrt{x}$. For the latter, we have to provide an enclosure of x, so we turn hypothesis `H` into one. We are now done building $\mathbf{y}$, so there is only $\mathbf{y} \in (-\infty; 4]$ left to prove:

```
I.subset
  (I.add 10%Z (I.fromZ 1) (I.sqrt 10%Z (I.bnd (F.fromZ 3) (F.fromZ 5))))
  (I.bnd F.nan (F.fromZ 4)) = true
```

The left-hand side of the equality is a closed Boolean expression made purely of interval operators. The `vm_compute` tactic uses the VM engine of Coq to normalize this expression to either `true` or `false`. Fortunately, it is the former, so the proof can be completed by using the reflexivity of equality.

Writing this kind of script is pointless. Indeed, the first tactic is hopelessly abstruse, while the remaining tactics simply follow the inductive structure of the expression to bound.

Moreover, if the enclosure bounds had not been plain integers, the abstruseness would have been even worse. That is why CoqInterval provides a reflection-based tactic named `interval` to perform this kind of formal proofs automatically. Section 3.4.1 presents the machinery used for reification and evaluation. Then, Section 3.4.2 shows the methods used by CoqInterval to fight the dependency effect.

3.4.1 Reification and evaluation

The tactic represents the expressions in the goal symbolically, as straight-line programs. Such a program is a standard way of encoding directed acyclic graphs and thus of explicitly sharing common subexpressions. It is just a list of statements indicating what the operations are and where their inputs can be found. The place where their output is stored is left implicit, as the result of an operation is always put at the top of the evaluation stack. The stack is initially filled with values corresponding to the constants of the program. Note that CoqInterval's evaluation model is simplistic, since the stack grows linearly with the size of the expression and no element of the stack is ever removed [Mel08b].

Let us illustrate this mechanism on an example. Consider $\int (t+\pi)\sqrt{t} - (t+\pi)\,dt$, a much simpler integral than the one of Equation (3.2). The straight-line program corresponding to its integrand is a list containing the operations to be performed. Each list item first indicates the arity of the operation, then the operation itself, and finally the depth at which the inputs of the operation can be found in the evaluation stack. Note that, in this example, t and π are seen as constants, so the initial stack contains values that correspond to these subterms. The only thing that will later distinguish the integration variable t from an actual constant such as π is that the value of t is initially at the top of the evaluation stack. The comments in the term below indicate the content of the stack before evaluating each item.

```
(*                              [t, pi] *)     Binary Add 0 1
(*                        [t+pi, t, pi] *)  :: Unary Sqrt 1
(*                [sqrt t, t+pi, t, pi] *)  :: Binary Mul 1 0
(*         [(t+pi)*sqrt t, sqrt t, ...] *)  :: Binary Sub 0 2
(* [(t+pi)*sqrt t - (t+pi), (t+pi)*...] *)  :: nil
```

Reification

To automate the proof, the tactic first needs to turn the expressions appearing in the goal into straight-line programs, *e.g.*, one program for the integrand and two programs for the integration bounds. This reification is performed using Ltac [Del00]. The most interesting construct of this meta-language is its ability to perform pattern matching on the syntax of terms. Moreover, if the code of the selected branch fails for some reason, then Coq will backtrack and restart using the next branch that matches.

The `list_find` function illustrates this construct. It returns the position of an element a in a list ℓ. It is used to compute the indices of the operands of a straight-line program.

```
Ltac list_find a l :=
  let rec aux l n :=
    match l with
    | nil        => false
    | cons a _   => n
```

```
    | cons _ ?l' => aux l' (S n)
    end in
  aux l 0.
```

Notice that, if a is found in ℓ, the returned term is n of type `nat` (constructors `0` and `S`). Otherwise, the returned term is *false* of type `bool`. That does not mean that Ltac is not a typed language. It just means that, as far as it is concerned, all the Coq terms have the same type `constr`, irrespective of their actual type. Thus, the function above has type `constr` → `constr` → `constr`.

Let us have a closer look at the second pattern: `(cons a _)`. It matches the term ℓ only if the head symbol of ℓ is the list constructor `cons` applied to a term that is syntactically equal to a. Notice the question mark in the third pattern; it means that ℓ' denotes a binder rather than an existing term.

The whole reification mechanism of CoqInterval is written in Ltac; at no point did I need to write an OCaml plugin for Coq.

Automation

The evaluation of a straight-line program depends on the interpretation of the arithmetic operations and on the values stored in the initial stack. For instance, if the arithmetic operations are the operations from the `Reals` library, *e.g.*, `Rplus`, and if the stack contains the symbolic value of the constants, then the result is the actual expression over real numbers.

Let us denote $[\![p]\!]_{\mathbb{R}}(\vec{x})$ the result of evaluating the straight-line program p with operators from `Reals` over an initial stack $\vec{x}$ of real numbers. Similarly, $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ denotes the result of evaluating p with interval operations over a stack of intervals. Then, thanks to the containment property, we can prove the following lemma once and for all.

Lemma 3.1 (Containment of evaluation) *For any program p, we have*

$$\forall \vec{x} \in \mathbb{R}^n,\ \forall \vec{\mathbf{x}} \in \mathbb{I}^n,\ (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\mathbb{R}}(\vec{x}) \in [\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}).$$

Lemma 3.1 is the basic block used by the `interval` tactic to prove enclosures of expressions. Given a goal $H_1, \ldots, H_n \vdash u \leq e \leq v$, the tactic first looks for a program p and a stack $\vec{x}$ of real numbers such that $[\![p]\!]_{\mathbb{R}}(\vec{x}) = e$. Note that this reification process cannot be proved to be correct, so Coq has to check that both sides of the equality are convertible. More precisely, the goal $u \leq e \leq v$ is convertible to $[\![p]\!]_{\mathbb{R}}(\vec{x}) \in [u; v]$ if u and v are floating-point numbers and if the tactic successfully reified the term. When the bounds are not floating-point numbers, they are also reified and evaluated using interval arithmetic, *e.g.*, $u \in \mathbf{u} = [\![p_u]\!]_{\mathbb{I}}(\vec{x})$, so that the original goal can be replaced by $[\![p]\!]_{\mathbb{R}}(\vec{x}) \in [\overline{u}; \underline{v}] \subseteq [u; v]$.

Once the tactic has reified the conclusion, it looks in the context for hypotheses H_k of the shape $u_k \leq x_i \leq v_k$ with x_i an element of the stack $\vec{x}$. That way, it can build a stack $\vec{\mathbf{x}}$ of intervals such that $\forall i,\ x_i \in \mathbf{x}_i$. If there is no such hypothesis, the tactic just uses $(-\infty; +\infty)$ for $\mathbf{x}_i$. The tactic can now apply Lemma 3.1 to replace the conclusion $[\![p]\!]_{\mathbb{R}}(\vec{x}) \in [\overline{u}; \underline{v}]$ by $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}) \subseteq [\overline{u}; \underline{v}]$. Finally, the tactic attempts to prove this new goal entirely by computation using the VM engine.

Integrability

Lemma 3.1 also implies, that if a function f can be reified as $t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x})$, then $\mathbf{t} \mapsto [\![p]\!]_{\mathbb{I}}(\mathbf{t}, \vec{\mathbf{x}})$ is an interval extension of f if $\forall i,\ x_i \in \mathbf{x}_i$. This will be important when dealing with integrals

in Section 3.5. Another matter when dealing with integrals is to prove that the integrand is actually integrable. We do so by using interval computations. This trick requires to explain the inner workings of the CoqInterval library in more detail. In particular, the library does not only provide bottom elements $\bot_{\mathbb{F}}$ and $\bot_{\mathbb{I}}$ for floating-point and interval arithmetic, it also provides some bottom element $\bot_{\mathbb{R}}$ for real arithmetic. In all that follows, $\overline{\mathbb{R}}$ denotes the set $\mathbb{R} \cup \{\bot_{\mathbb{R}}\}$ of *extended real* numbers. The alternate scheme $[\![p]\!]_{\overline{\mathbb{R}}}$ produces the value $\bot_{\mathbb{R}}$ as soon as an operation is applied to inputs that are outside the usual definition domain of the operator. For instance, the result of dividing one by zero in $\overline{\mathbb{R}}$ is $\bot_{\mathbb{R}}$, while it is unspecified in $\mathbb{R}$. This $\bot_{\mathbb{R}}$ element is then propagated along the subsequent operations. Thus, the following equality holds, using the trivial embedding from $\mathbb{R}$ into $\overline{\mathbb{R}}$.

Lemma 3.2 (Evaluation using extended reals) *For any program p, we have*

$$\forall \vec{x} \in \mathbb{R}^n,\ [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}) \neq \bot_{\mathbb{R}} \Rightarrow [\![p]\!]_{\mathbb{R}}(\vec{x}) = [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}).$$

Moreover, since both elements $\bot_{\mathbb{R}}$ and $\bot_{\mathbb{I}}$ are propagated along computations, by extending the definition of an enclosure so that $\bot_{\mathbb{R}} \in \bot_{\mathbb{I}}$ holds, we can prove a variation of Lemma 3.1.

Lemma 3.3 (Containment and extended reals) *For any program p, we have*

$$\forall \vec{x} \in \overline{\mathbb{R}}^n,\ \forall \vec{\mathbf{x}} \in \mathbb{I}^n,\ (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}) \in [\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}).$$

In CoqInterval, Lemma 3.1 is deduced from both Lemmas 3.2 and 3.3, thanks to two other properties of $\bot_{\mathbb{I}}$. First, $(-\infty; +\infty) \subseteq \bot_{\mathbb{I}}$ holds. So, the consequent of Lemma 3.3 trivially holds when $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ evaluates to $\bot_{\mathbb{I}}$. Second, $\bot_{\mathbb{I}}$ is the only enclosure of $\bot_{\mathbb{R}}$. So, if $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ does not evaluate to $\bot_{\mathbb{I}}$, then the antecedent of Lemma 3.2 holds.

Let us go back to the issue of proving integrability. By definition, whenever $[\![p]\!]_{\overline{\mathbb{R}}}(\vec{x})$ does not evaluate to $\bot_{\mathbb{R}}$, the inputs $\vec{x}$ are part of the definition domain of the expression represented by p. But we can actually prove a stronger property for some programs: not only is $\vec{x}$ part of the definition domain, it is also part of the continuity domain. Currently, the only programs for which we are not able to prove this property are those computing an integer part (but there might be more incompatible operators in the future). This gives the following lemma.

Lemma 3.4 (Continuity) *For any program p that does not contain any integer part, given a point $t_0 \in \mathbb{R}$ and a stack $\vec{x} \in \mathbb{R}^n$, if $[\![p]\!]_{\overline{\mathbb{R}}}(t_0, \vec{x}) \neq \bot_{\mathbb{R}}$ holds, then $t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x})$ is continuous at point t_0.*

By combining Lemmas 3.1 and 3.4, we obtain a numeric/symbolic method to prove that a function is continuous on a domain. Indeed, we just have to compute an enclosure of the function on that domain, and to check that it is not $\bot_{\mathbb{I}}$. A closer look at the way naive integral enclosures are computed provides the following corollary: whenever the enclosure of the integral is not $\bot_{\mathbb{I}}$, the function is actually continuous and thus integrable on any compact of the input domain. This solves the issue for proper integrals.

3.4.2 Higher-order methods

As mentioned in Section 2.2.3, the main trouble with naive interval arithmetic is the dependency effect, as it cannot track correlated variations between subexpressions. A simple way to mitigate this issue is bisection: if a property cannot be proved by interval arithmetic on the whole domain of a variable, split this domain into smaller subdomains and try again. Note that, since the goal has been reified, it is especially easy to restart an interval computation as many times as needed.

Let us illustrate this approach with the example of Section 2.2.3. The objective was to bound $\tilde{y} - y = (1 + x + x^2/2) - \exp x$ over $x \in [0; 1]$. Using naive interval arithmetic, we cannot obtain a better enclosure than $|\tilde{y} - y| \leq 1.72$.

```
Variable x: R.
Hypothesis H: 0 <= x <= 1.

Goal Rabs (1 + x + x*x/2 - exp x) <= 172/100.
Proof. interval. Qed. (* OK *)

Goal Rabs (1 + x + x*x/2 - exp x) <= 171/100.
Proof. Fail interval. Abort. (* failure *)
```

Using bisection, we can get to the almost optimal bound $|\tilde{y} - y| \leq 0.22$. The `(i_bisect x)` option tells the tactic to recursively split an enclosure of x in two whenever it fails to prove the property. The `(i_depth 12)` option ensures termination by making the tactic give up if the depth of bisection reaches 12. Here, it means that subdomains of width 2^{-11} are the smallest to be considered.

```
Goal Rabs (1 + x + x*x/2 - exp x) <= 22/100.
Proof. interval with (i_bisect x, i_depth 12). Qed.
```

On this example, the tactic has to split the domain of x into 31 subdivisions. The smallest subdomains, *i.e.*, the ones of width 2^{-11}, are on the right side of $[0; 1]$, where the subexpressions vary the fastest. Since the successful subdomains can be seen as the leaves of a binary tree, it means that 61 enclosures of x were tested to fully verify the property. While Coq's VM engine is sufficiently efficient for that to happen in a split second, it is easy to build examples where the number of subdivisions is so large that the tactic would never succeed.

As explained, the smallest subdivisions are needed where subexpressions vary the fastest, so we somehow have to take into account these variations. One way to do so is to use Taylor-Lagrange's formula. At order 0, it is also known as the *mean-value* theorem. For $x_0 \in \mathbf{x}$ and under some assumptions on f, it looks as follows:

$$\forall x \in \mathbf{x},\ \exists \xi \in \mathbf{x},\ f(x) = f(x_0) + (x - x_0) \cdot f'(\xi).$$

By weakening it using the containment property, this formula becomes

$$\forall x \in \mathbf{x},\ f(x) \in \mathbf{f}([x_0; x_0]) + (\mathbf{x} - x_0) \cdot \mathbf{f}'(\mathbf{x}).$$

This time, the interval extension of f is used on a point interval, so the dependency effect is almost nonexistent. But we now have to evaluate the interval extension of the derivative f'. The dependency effect for $\mathbf{f}'$ might be worse than for $\mathbf{f}$, but the hope is that the factor $\mathbf{x} - x_0$

can keep it in check. Indeed, even if $(\mathbf{x} - x_0) \cdot \mathbf{f}'(\mathbf{x})$ has possibly a larger width than $\mathbf{f}(\mathbf{x})$ initially, this width should decrease much faster once $\mathbf{x}$ is split using bisection.

One way to obtain $\mathbf{f}'$ is to symbolically compute f' and then turn into an interval extension using the containment property. Marc Daumas, César Muñoz, and I, used this approach to verify an enclosure with the PVS proof assistant [DMM05].

By making use of CoqInterval's reification and evaluation of expressions, there is another way to compute $\mathbf{f}'(\mathbf{x})$. We just have to define a set of arithmetic operators that can compute it. More precisely, we define a set of operators that compute both $\mathbf{f}(\mathbf{x})$ and $\mathbf{f}'(\mathbf{x})$ simultaneously, as is done in *automatic differentiation* [BKSF59]. Here are a few examples:

$$\begin{aligned}(\mathbf{u}, \mathbf{u}') + (\mathbf{v}, \mathbf{v}') &= (\mathbf{u} + \mathbf{v}, \mathbf{u}' + \mathbf{v}'),\\ (\mathbf{u}, \mathbf{u}') \times (\mathbf{v}, \mathbf{v}') &= (\mathbf{u} \cdot \mathbf{v}, \mathbf{u}' \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{v}'),\\ \exp(\mathbf{u}, \mathbf{u}') &= (\exp(\mathbf{u}), \mathbf{u}' \cdot \exp(\mathbf{u})).\end{aligned}$$

As with naive interval arithmetic, there is a containment property satisfied by these pairs of intervals. It is preserved by composition, so we get $(\mathbf{f}(\mathbf{x}), \mathbf{f}'(\mathbf{x}))$ at the end of the computation. Once we have $\mathbf{f}'(\mathbf{x})$, we compute $\mathbf{f}([x_0; x_0])$ and we apply the interval version of Taylor-Lagrange's formula. The first member of the pair could be discarded at that point, but we use it to further refine the result:

$$f(x) \in \mathbf{f}(\mathbf{x}) \cap (\mathbf{f}([x_0; x_0]) + (\mathbf{x} - x_0) \cdot \mathbf{f}'(\mathbf{x})).$$

In some cases, we can do slightly better. If the interval $\mathbf{f}'(\mathbf{x})$ happens to be of constant sign, then we can prove that f is monotone on $\mathbf{x}$. In that case, we can just compute $\mathbf{f}([\underline{x}; \underline{x}])$ and $\mathbf{f}([\overline{x}; \overline{x}])$, take their convex hull, and thus obtain a superset of $f(\mathbf{x})$. Since the input intervals are point intervals, the dependency effect is minimal and the enclosure is tight. As a consequence, when combining automatic differentiation with bisection, no more splitting is needed once the interval evaluation $\mathbf{f}'$ has constant sign on each of the subintervals [Mel08b].

Automatic differentiation is enabled by the `(i_bisect_diff x)` option. It tells the tactic both to derive with respect to x and to perform a bisection on the domain of x. This gives the following script.

```
Goal Rabs (1 + x + x*x/2 - exp x) <= 22/100.
Proof. interval with (i_bisect_diff x, i_depth 3). Qed.
```

Using naive interval arithmetic, the tactic needed to study 61 enclosures of x. Using automatic differentiation, only 5 enclosures are needed, since the successful subintervals are $[0; 0.5]$, $[0.5; 0.75]$, and $[0.75; 1]$, with the latter making use of monotony. The time needed for each subinterval, however, has grown. Let us look at the number of times the exp function is evaluated, as it is the bottleneck of the evaluation. For naive interval arithmetic, there are 122 calls to exp. For automatic differentiation, there are 22 calls.

The improvement is noticeable. But in some cases, it is still not enough, as the example of Section 4.1 will show. For automatic differentiation, we have used Taylor-Lagrange's formula at order 0. But we could go to a higher order n:

$$\forall x \in \mathbf{x},\ \exists \xi \in \mathbf{x},\ f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}.$$

Turning this equality into an interval enclosure does not work that well, since it would contain numerous factors $(\mathbf{x} - x_0)^k$, whose correlation would be lost. So, it is better to see

Taylor-Lagrange's formula as providing a Taylor expansion, *i.e.*, a polynomial in $x - x_0$ and a remainder. By manipulating polynomials, we keep track of the dependencies till the end.

Let us recapitulate. With naive interval arithmetic, enclosures were represented by a single interval. With automatic differentiation, they were represented by a pair of intervals. Now they are represented by a pair $(\vec{\mathbf{p}}, \mathbf{\Delta})$ of a vector $\vec{\mathbf{p}}$ of intervals and an interval $\mathbf{\Delta}$. More precisely, given a point $x_0 \in \mathbf{x}$, the pair $(\vec{\mathbf{p}}, \mathbf{\Delta})$ represents an enclosure of f over $\mathbf{x}$ if there exists a polynomial p with real coefficients $p_i \in \mathbf{p}_i$ such that

$$\forall x \in \mathbf{x},\ f(x) - p(x - x_0) \in \mathbf{\Delta}.$$

As with naive interval arithmetic and automatic differentiation, we then define an arithmetic over this new kind of enclosures. Let us assume that $(\vec{\mathbf{p}}_f, \mathbf{\Delta}_f)$ and $(\vec{\mathbf{p}}_g, \mathbf{\Delta}_g)$ enclose the functions f and g over the interval $\mathbf{x}$ using point x_0. Let us also assume that $\vec{\mathbf{p}}_f$ and $\vec{\mathbf{p}}_g$ have the same size s for simplicity. The enclosures of f and g mean that there are some coefficients p_{fi} and p_{gi} such that

$$\forall i < s,\ p_{fi} \in \mathbf{p}_{fi} \wedge p_{gi} \in \mathbf{p}_{gi},$$

$$\forall x \in \mathbf{x},\ \begin{cases} f(x) - \sum_{i<s} p_{fi} \cdot (x - x_0)^i & \in \mathbf{\Delta}_f, \\ g(x) - \sum_{i<s} p_{gi} \cdot (x - x_0)^i & \in \mathbf{\Delta}_g. \end{cases}$$

By summing these two enclosures, we deduce that the polynomial $p = p_f + p_g$ satisfies

$$\forall x \in \mathbf{x}, \quad (f(x) + g(x)) - p(x - x_0) \in \mathbf{\Delta}_f + \mathbf{\Delta}_g.$$

Therefore, $(\vec{\mathbf{p}}_f + \vec{\mathbf{p}}_g, \mathbf{\Delta}_f + \mathbf{\Delta}_g)$ is an enclosure of $f + g$ over the interval $\mathbf{x}$ using point x_0. The formula is similar for subtraction. For multiplication, the straightforward approach would produce a polynomial of size $2s$. To avoid polynomials growing in size (and thus subsequent arithmetic operations getting costlier), the result of a multiplication is usually truncated to a polynomial approximation of size s by enlarging the error interval accordingly. For an elementary function, one would compute the Taylor series of the function, compose it with the input polynomial, and truncate it to get an enclosure. So, these polynomial approximations are also named *Taylor models*.

This Coq development originated from the CoqApprox working group of the TaMaDi project.[10] The motivation was not to automate the verification of enclosures, but to produce rigorous polynomial approximations that could help with matters related to the *table-maker dilemma* [Jol11, BJMD+12]. This development was later integrated into CoqInterval by Érik Martin-Dorel and me [MDM16].

The following script uses this feature. It is enabled by the option `(i_bisect_taylor x 2)`, which tells the `interval` tactic to use polynomial approximations of degree 2 and of variable x, and to bisect along x.

```
Goal Rabs (1 + x + x*x/2 - exp x) <= 22/100.
Proof. interval with (i_bisect_taylor x 2, i_depth 1). Qed.
```

Notice that bisection is no longer needed. Indeed, the polynomial enclosure of the whole expression (except the absolute value) over $[0; 1]$ using point 0.5 is roughly

$$-0.024 - 0.15t - 0.32t^2 + [-0.039; 0.030],$$

[10] `http://tamadi.gforge.inria.fr/`

which has properly captured the variations of the subexpressions, as can be seen from the small error interval. The number of floating-point evaluations of exp is now 10. Note that this cost does not depend on the degree of the polynomial. Indeed, since exp satisfies a linear differential equation, the coefficients of its Taylor model satisfies a linear recurrence and exp is only evaluated to initialize this recurrence [MDMP+13]. So, we could have used a much larger degree without incurring much of an overhead. For this example, it does not make sense to go higher than degree 10, though, since the round-off errors then become prevalent, due to the default floating-point precision of 30 bits.

Finally, note that the approach based on Taylor models does not completely supersede the one based on automatic differentiation. Indeed, in some corner cases, the latter might benefit from its analysis of monotony. For example, the tactic is able to prove $\forall x \in \mathbb{R},\ 1 + x \leq \exp x$ using automatic differentiation. Indeed, after splitting the input interval $(-\infty; +\infty)$ into two, it notices that $1 + x - \exp x$ has a nonnegative derivative on $(-\infty; 0]$ and a nonpositive one on $[0; +\infty)$, using interval arithmetic. Thus, $1 + x - \exp x$ reaches its maximum at $x = 0$, which ensures $1 + x - \exp x \leq 1 + 0 - \exp 0 = 0$.

3.4.3 Assessment

Backward propagation

A deficiency of the tactic is that it operates in a purely forward manner. In particular, it does not take into account preexisting enclosures of intermediate expressions, which prevents it from verifying a property like $\forall x \forall y,\ x^2 + y^2 = 1 \Rightarrow -1 \leq x \leq 1$. To do so, one needs to implement *backward propagators* [JKDW01, §4.2]. Naive interval arithmetic is a forward propagator, as it tells you how to compute an enclosure of $f(\vec{x})$ given the enclosures $\vec{x} \in \vec{\mathbf{x}}$. A backward propagator tells you how to refine the intervals $\vec{\mathbf{x}}$ given an enclosure of $f(\vec{x})$. For example, given three enclosures $u \in \mathbf{u}$, $v \in \mathbf{v}$, and $u + v \in \mathbf{w}$, then $\mathbf{u} \cap (\mathbf{w} - \mathbf{v})$ is a refined enclosure of u. Once we have both forward and backward propagators, there is no longer an explicit flow of intervals. So, as a side effect, the tactic could be made to handle arbitrary Boolean propositions involving enclosures. The evaluation model of CoqInterval, however, is poorly suited for that task, since it is built around forward propagation.

Polynomial approximation

The terminology "Taylor model" comes from the use of Taylor expansions to compute the polynomial enclosure that results from the application of an elementary function. Taylor expansions provide good polynomial approximations, but only around the expansion point x_0. Thus, the computed error intervals might be much larger than what is theoretically possible to achieve. In turn, this means that bisection might need many more subintervals to conclude. Rather than using the polynomial coming from truncating the Taylor expansion of an elementary function, it might be better to interpolate this elementary function at Chebyshev's nodes [BJ10].

Multivariate goals

Another deficiency is that, as can be guessed from the options supported by the tactic, CoqInterval is designed to verify univariate goals. More precisely, expressions can use several variables, but the user will only be able to mitigate the loss of correlation due to one of them.

Note that, if only one of the variables occurs with several occurrences, then this limitation does not matter. Similarly, if the user can split the target expression into several univariate subexpressions, *e.g.*, $f(g(x), h(y))$, then the limitation can be lifted by performing multiple calls to the tactic [MDM16]. In the general multivariate case, however, the tactic falls short. At the very least, the tactic should be improved so that it can bisect along several variables, *e.g.*, in a round-robin fashion. With respect to bisection, another missing feature is feedback, so as to tell the user why the tactic failed to prove a property. For example, the bisection algorithm could be instrumented to return the failing subdomain.

Adding multivariate bisection makes it possible to tackle a much larger families of goals, but it certainly does not help with the time it takes to do so. Indeed, if a bisection of depth d is needed in the univariate case, one might need a bisection of depth $n \cdot d$ in the n-variable case, hence furthering the exponential blowup. Thus, some multivariate methods need to be devised to counter the dependency effect. First of all, the implementation of automatic differentiation should be generalized to the multivariate case. Instead of propagating the derivative with respect to a single input variable, the tactic would propagate the partial derivatives with respect to each input variables. The time complexity of automatic differentiation in the univariate case was $O(\ell)$ with ℓ the length of the straight-line program; it is now $O(n \cdot \ell)$. To reduce this complexity back to $O(n)$ (assuming only one expression needs to be enclosed at the end), one might instead consider *backward* propagating the partial derivatives of the result with respect to the intermediate subexpressions [Spe80]. Note that, as with backward propagators, CoqInterval's evaluation model is not suitable for the *reverse mode* of automatic differentiation.

Another way to preserve first-order correlations is *affine arithmetic* [CS93]. A set of noise symbols $(\varepsilon_i)_i$ that satisfy $\varepsilon_i \in [-1; 1]$ is introduced. Then, each subexpression can be enclosed using an affine combination of these noise symbols: $e = c + \sum_i \alpha_i \cdot \varepsilon_i$. To do so, each input variable has one dedicated noise symbol, while intermediate expressions use a fresh noise symbol to account for second-order effects. At the end, one just has to turn the affine combination representing the target expression into an enclosure. Affine arithmetic is especially well suited to the evaluation model of CoqInterval, since it is straightforward to associate fresh noise symbols with the depth of the evaluation stack. The main difficulty with affine arithmetic is that fresh noise symbols constantly appearing might cause a time complexity of $O(\ell^3)$. Thus, it is important to forcefully remove the ones that seem useless, but that might be difficult given CoqInterval's evaluation model.

A few words should be said about the extension of polynomial enclosures to the multivariate case. In the univariate case, one could expect a time complexity of $O(\ell \cdot s^2)$, with s the degree of the polynomials. In the multivariate case, if we assume the same maximal degree for each of the input variables, the complexity is more like $O(\ell \cdot s^{2n})$, which is impractical. That does not mean that multivariate polynomial approximations are useless. It just means that the we have to be especially careful when choosing the polynomial basis. For instance, we might decide that only one input variable can have the maximal degree and that all the other ones should be limited to degree 0 or 1. We could limit it further by saying that, for each monomial, only one other variable can appear. A variation of the latter would be to go back to plain univariate polynomials, but instead of using constant coefficients, we would represent the coefficients using affine arithmetic to account for all the other variables. Note that, in all these limited bases, one variable has to be singled out as the main variable, while the others are handled in a degraded way. Fortunately, some use cases are hardly impeded

by this limitation. For instance, in the case of definite univariate integrals, the integration variable plays a prevalent role; and so does it for ordinary differential equations.

Polynomial evaluation

Finally, there is an issue that was kept quiet in Section 3.4.2. Since we need to bound expressions, how do we get from a polynomial enclosure to an interval enclosure? The solution that immediately comes to mind is to perform an interval evaluation of the polynomial, *e.g.*, using Horner's scheme. Unfortunately, the dependency effect strikes again [Sta95]. This can be mitigated using some improved evaluation of polynomials [CG02]. We did not formalize the whole method in CoqInterval though, so only the quadratic part of the polynomials is improved:

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot \ldots)) = a_0 - \frac{a_1^2}{4a_2} + a_2 \cdot \left(x + \frac{a_1}{2a_2}\right)^2 + x^3 \cdot (a_3 + x \cdot (a_4 + x \cdot \ldots)).$$

There have been numerous other attempts at bounding univariate polynomials in a formal setting. In order to locate the extrema of some polynomial P, Harrison originally computed small intervals enclosing the roots of the derivative P' using its Sturm sequence [Har97b]. Harrison also considered using a sum-of-square decomposition for proving properties of polynomial systems [Har07], which is a problem much more general than simply bounding P. The decomposition $\sum R_i^2$ of a polynomial $R = P + a$ is especially well suited for formal proofs. Indeed, while the decomposition may be difficult to obtain, the algorithm does not have to be formally proved. Only the equality $R = \sum R_i^2$ has to, which is easy and fast. A straightforward corollary of this equality is the property $P \geq -a$.

However, both approaches seem to be less efficient than doing a variation analysis of Q in HOL Light [Har00a]. Again, this analysis looks for the roots of the derivative P', but instead of using Sturm sequences to obtain the enclosures of these roots, Harrison recursively enclosed all the roots of its derivatives. Indeed, a polynomial is monotonic between two consecutive roots of its derivative. The recursion stops on linear polynomials, which have trivial roots.

Another approach is the representation of P' in Bernstein's polynomial basis. Indeed, if the number of sign changes in the sequence of coefficients is zero or one, then it is also the number of roots (similar to Descartes' Law of Signs). If it is bigger, then De Casteljau's algorithm can be used to efficiently compute the Bernstein representations of P' on two subintervals. A bisection can therefore isolate all the extrema of P. Zumkeller implemented and proved this method in Coq [Zum06]. Muñoz and Narkawicz implemented a similar approach in PVS [MN13].

Finding the good tradeoff between performance and tightness of the polynomial evaluation is important, since there is not much point in slowly computing a tight polynomial enclosure (characterized by a small error interval) if the dependency effect prevents us from computing tight bounds on the polynomial part.

3.5 Quadrature

We now have enough components to automatically and formally verify enclosures of definite integrals. Assia Mahboubi, Thomas Sibut-Pinote, and I, integrated it to the CoqInterval library [MMSP16, MMSP19]. Section 3.5.1 considers the case of proper integrals. Section 3.5.2 then extends it to the improper case.

3.5.1 Proper definite integrals

Let us start with the case of proper integrals, *i.e.*, performing the quadrature $\int_u^v f$ with u and v some finite real numbers and f Riemann-integrable over $[\min(u,v);\max(u,v)]$. This integration domain will be denoted as $[u;v]$ in the following, even when $u \geq v$. We denote $\mathbf{f}$ an interval extension of f over $[u;v]$. We denote $\mathbf{u}$ and $\mathbf{v}$ some intervals enclosing the bounds.

Naive enclosures

Since function f is enclosed between the constant functions $x \mapsto \inf_{[u;v]} f$ and $x \mapsto \sup_{[u;v]} f$, its integral is enclosed between the integrals of these two constant functions. This leads to the following lemma, as well as its corollary about interval extensions.

Lemma 3.5

$$\int_u^v f \in (v-u) \cdot \mathrm{hull}\{f(t) \mid t \in [u;v]\}.$$

Corollary 3.6

$$\int_u^v f \in (\mathbf{v}-\mathbf{u}) \cdot \mathbf{f}(\mathrm{hull}(\mathbf{u},\mathbf{v})).$$

This approach is straightforward to implement but it produces poor enclosures of the integral when f is not close to a constant function. For example, the integral $\int_0^1 x^2\,dx = \frac{1}{3}$ will be enclosed into $(1-0)\cdot[0;1] = [0;1]$. Note that this imprecision is not due to the dependency effect, since $[0;1]$ is the optimal enclosure of x^2 over $[0;1]$.

Yet, as with naive interval arithmetic, the use of bisection can help improving the enclosure. By splitting the integration domain into two parts, we get the following enclosure:

$$\int_0^1 x^2\,dx = \int_0^{0.5} x^2\,dx + \int_{0.5}^1 x^2\,dx \in (0.5-0)\cdot[0;0.25] + (1-0.5)\cdot[0.25;1] = [0.125;0.625].$$

This example shows that splitting the integration domain into two parts divides the width of the enclosure by two, but the amount of computations is doubled. Thus, as with the dependency effect, the use of bisection for quadrature should be reserved for corner cases where faster methods do not apply.

Polynomial approximation

Let us detail the non-naive method we formalized in CoqInterval [MMSP16]. It is based on the Taylor models that CoqInterval generates. Indeed, if the polynomial p approximates f, then $f-p$ is close to a constant function, so even a naive method can give a tight enclosure of $\int_u^v (f-p)$. Using the enclosures of Section 3.4.2, we get the following lemma.

Lemma 3.7 *Let f be approximated over $[u;v]$ by $p \in \mathbb{R}[X]$ and $\mathbf{\Delta} \in \mathbb{I}$ in the sense that*

$$\forall x \in [u;v],\ f(x) - p(x) \in \mathbf{\Delta}.$$

Then for any primitive P of p, we have

$$\int_u^v f \in P(v) - P(u) + (v-u)\cdot\mathbf{\Delta}.$$

Integrability

Up to now, we have assumed that f was Riemann-integrable over $[u;v]$. For the sake of automation, it would be better if the user had not to prove it manually. So, we want the proof of integrability to be a byproduct of the computation of the integral. In other words, can we formally prove that, if one of the above methods returns a bounded interval, then not only is this interval an enclosure of the integral, but it is also a proof that the function was integrable?

As explained in Section 3.4.1, we can. Indeed, except for the integer part, all the functions supported by CoqInterval are continuous on their definition domain, and their interval extensions only return an actual interval (*i.e.*, not $\bot_{\mathbb{I}}$) if the input interval lies in the definition domain. As a consequence, we can improve the statement of Corollary 3.6 as follows.

Lemma 3.8 *If f is expressible in CoqInterval, if it does not use any integer part, and if $\mathbf{f}(\mathrm{hull}(\mathbf{u},\mathbf{v}))$ evaluates to a bounded interval, then f is integrable over $[u;v]$ and we have*

$$\int_u^v f \in (\mathbf{v}-\mathbf{u})\cdot\mathbf{f}(\mathrm{hull}(\mathbf{u},\mathbf{v})).$$

Testing that the integrand does not rely on the integer part can be done by visiting the reified object that represents it.

3.5.2 Improper definite integrals

An improper integral can be seen as the sum of a proper integral and a remainder. The proper part can be bounded accordingly to the methods described in Section 3.5.1. So, let us focus on the remainder.

We only consider improper integrals of the shape $\int_u^v fg$ where either $u = 0^+$ or $v = +\infty$, and f is bounded. As for g, it belongs to a catalog of functions with known enclosures of their integral, such as $x^\alpha \log^\beta x$. In the following, we only present the case $v = +\infty$; refer to the journal article for the case $u = 0^+$ [MMSP19]. To handle the fg integrand, we use the following lemmas.

Lemma 3.9 *Let $f, g : \mathbb{R} \to \mathbb{R}$. Suppose that, over $[u;+\infty)$, f is bounded, f and g are continuous, and g has a constant sign. Moreover, suppose that $\int_u^{+\infty} g$ exists. Then $\int_u^{+\infty} fg$ exists, and*

$$\int_u^{+\infty} fg \in \mathrm{hull}\{f(t) \mid t \geq u\} \cdot \int_u^{+\infty} g.$$

Corollary 3.10 *Let $\mathbf{f}$ be an interval extension of f. Let $\mathbf{G}$ be an interval extension of the improper integral $x \mapsto \int_x^{+\infty} g$. For any enclosure $u \in \mathbf{u}$, we have*

$$\int_u^{+\infty} fg \in \mathbf{f}(\mathrm{hull}(\mathbf{u},+\infty)) \cdot \mathbf{G}(\mathbf{u}).$$

In order to use this corollary, we need to find a suitable extension $\mathbf{G}$ for the remainder of the integral of g. So, we look at two classes of well-known integrable functions. For the positive function $g(x) = e^{\gamma x}$ with $\gamma < 0$, we use the fact that

$$\int_u^{+\infty} e^{\gamma x}\,dx = -\frac{e^{\gamma u}}{\gamma}.$$

Let us now consider the family of functions $g(x) = x^\alpha \log^\beta x$ with $\alpha, \beta \in \mathbb{R}$. These functions are of constant positive sign on $[1; +\infty)$. They are integrable at $+\infty$ only when $\alpha < -1$, or when $\alpha = -1$ and $\beta < -1$. The interesting cases are as follows:

1. When $\alpha = -1$ and $\beta < -1$, we have a closed formula:
$$\int_u^{+\infty} \frac{\log^\beta x}{x}\,dx = -\frac{\log^{\beta+1} u}{\beta+1}.$$
2. When $\alpha < -1$ and $\beta = 0$, we have a closed formula:
$$\int_u^{+\infty} x^\alpha\,dx = -\frac{u^{\alpha+1}}{\alpha+1}. \tag{3.3}$$
3. When $\alpha < -1$ and $\beta < 0$, there is no closed formula, but by moving $\log^\beta x$ into the bounded part of Lemma 3.10, we go back to the previous case.
4. When $\alpha < -1$ and $\beta \in \mathbb{N}$, we have a closed formula, obtained by recurrence on β using Equation (3.3) and by integrating by parts:
$$\int_u^{+\infty} x^\alpha \log^\beta x\,dx = -\left(\frac{u^{\alpha+1}\log^\beta u}{\alpha+1}\right) - \frac{\beta}{\alpha+1}\int_u^{+\infty} x^\alpha \log^{\beta-1} x\,dx.$$
For example, we get
$$\int_1^{+\infty} \frac{\log x}{x^2}\,dx = -\left(\frac{1^{-1}\log^1 1}{-1}\right) - \frac{1}{-1}\int_1^{+\infty}\frac{dx}{x^2} = 0 - \frac{1^{-1}}{-1} = 1.$$

3.5.3 Some examples

We can now have a look at Equation (3.2), our original example:
$$\int_{-\infty}^{+\infty} \frac{(0.5\cdot\log(\tau^2+2.25)+4.1396+\log\pi)^2}{0.25+\tau^2}\,d\tau.$$

We denote h its integrand. Note that this integral does not fit into the setting of Section 3.5.2, so we have to do a bit of preprocessing. First of all, we split the integration domain into three parts: $(-\infty; -100{,}000] \cup [-100{,}000; 100{,}000] \cup [100{,}000; +\infty)$, as was done by Helfgott [Hel14].

The center part is a proper integral, so Section 3.5.1 applies. Using Taylor models of degree 10 (the default value), at most 20 levels of domain splitting, a target enclosure width of 2^{-10}, and a floating-point precision of 40 bits, it takes less than 20 seconds for Coq to obtain and verify a proof of the following enclosure:
$$\int_{-100{,}000}^{100{,}000} h \in [226.8429; 226.8436].$$

Let us look at the remainders now. Note that, thanks to the parity of the function, we need to consider only one of them, *e.g.*, the positive domain. Since this is a remainder, we

have to express h as fg with f bounded and g well-known, which leads to the following improper integral:

$$\int_{100{,}000}^{+\infty} \frac{1 + \left(\frac{0.5 \cdot \log(1+2.25/\tau^2) + 4.1396 + \log \pi}{\log \tau}\right)^2}{1 + 0.25/\tau^2} \cdot \frac{\log^2 \tau}{\tau^2}\, d\tau.$$

Proving that both old and new integrands are equal over $[100{,}000; +\infty)$ is the matter of ten lines of Coq script. Directly computing the remainder does not give a tight enough enclosure, so we let the `interval` tactic subdivide the integration domain into subintervals. Since the upper bound is infinite, the tactic chooses the subintervals by splitting at twice the lower bound. We ask for at most 4 levels of domain splitting, while other parameters are left to their default values. It takes less than one second for Coq to verify the following enclosure:

$$2 \cdot \int_{100{,}000}^{+\infty} h \in [0.0061; 0.0064].$$

As a consequence, we have a formal proof that the integral over the whole domain is enclosed in $[226.849; 226.850]$. In particular, this means that Helfgott's upper bound (226.844) was underestimated. Presumably, he got the correct enclosure for the proper part, using VNODE-LP [Ned06], but he underestimated the improper parts by a factor 10.

When Helfgott was looking for a reliable tool, he was not experimenting with the above integral but with the following one:

$$\int_0^1 \left|\left(x^4 + 10x^3 + 19x^2 - 6x - 6\right)\, e^x\right| dx \simeq 11.147{,}310{,}550{,}057{,}139{,}734.$$

This integral is interesting because it has a closed formula (using algebraic numbers and Napier's constant), but the absolute value makes the derivative of the integrand discontinuous and thus causes troubles for quadrature methods. For example, the `quad` and `quadgk` methods of Mathlab/Octave computes only 10 correct decimal digits out of the 15 expected ones, yet they do not emit any warning message. As for the `verifyquad` method of INTLAB [Rum99], it would return an enclosure that does not contain the exact value, since its algorithm is reliable only if the integrand is four times differentiable. Since then, the bug in `verifyquad` has been fixed by disabling support for absolute values in integrands. Note that absolute value is not supported by VNODE-LP either.

Even if the mentioned tools return an incorrect result, they do so near instantly. So, it is interesting to look at the time it takes for CoqInterval to compute a formally verified enclosure. The following table shows, for various choices of absolute errors, how many seconds it took and which options were passed to the tactic (absolute width of integral enclosures, degree of Taylor models for integrands, depth of bisection for the integration domain, and binary floating-point precision).

Error	Time	Width	Degree	Depth	Precision
10^{-3}	0.3	2^{-10}	5	8	30
10^{-6}	0.5	2^{-20}	7	13	40
10^{-9}	0.7	2^{-30}	9	18	50
10^{-12}	1.0	2^{-40}	11	22	60
10^{-15}	1.4	2^{-50}	12	28	70
10^{-18}	1.8	2^{-60}	13	34	80

Many more details and examples can be found from our article, as well as some considerations about the time complexity of our approach [MMSP19].

3.5.4 Assessment

Taylor models

Though we were originally not aware of it, the use of Taylor models for enclosing proper definite integrals had already been experimented [CR87]. The basic idea is the same, but there are several subtle improvements in their approach. First of all, they infer tighter enclosures of their polynomial approximations. Indeed, for f approximated by $(\mathbf{p}, \mathbf{\Delta})$ over $[u; v]$, the only property we are using is $\forall t \in [u; v],\ f(t) - p(t) \in \Delta$. But Corliss and Rall are making use of two other properties. First, p is computed at the midpoint m of $[u; v]$, which means that, to compute $\int_u^v p(t - m)\, dt = P(v - m) - P(u - m)$, monomials of p of odd degrees can be ignored, which speeds up computations by a factor two and reduces the width of enclosures due to round-off errors. Second, $\mathbf{\Delta}$ is not just the absolute error between f and p, it is in fact an enclosure of $f^{(n)}(t)$ over $[u; v]$, up to some factor $(v-u)^n/(2^n n!)$. So, the width of the integral enclosure is much less than $(v - u) \cdot \mathbf{\Delta}$, since we have some knowledge on the n-th derivative of f. Transposing this last property into CoqInterval would require us to revisit the whole formalization of Taylor models (see also Section 3.4.3).

Another shortcoming of CoqInterval is that the user has to guess which polynomial degree is best to reach a target accuracy. Moreover, this chosen degree will be applied to all the subintervals of the integration domain, even those where Taylor models are ill-behaved. The approach of Corliss and Rall does not suffer from this issue, since they dynamically select the best degree. Indeed, they are not constrained by a formal system, so they have access to a clock. In particular, they can measure how long computing a Taylor model takes, and thus they can decide if increasing the degree is worth it or if it would be better to split the integration domain. Note that, even with an oracle, it would be difficult to reproduce such a feature, since the oracle would have to know in advance how long it would take for Coq to perform some given computation.

Gaussian quadrature

Regarding proper definite integrals, there is a completely different approach that would be worth experimenting in a formal setting. It was implemented by Johansson in the ARB library [Joh18]. It is based on the following inequality:

$$\left| \int_{-1}^{1} f - \sum_{k=0}^{n} w_k \cdot f(x_k) \right| \leq M \frac{5\rho^{-2n}}{\rho^2 - 1}$$

with f analytic on an ellipse of foci ± 1, with ρ the sum of the semi-axes of this ellipse, and with M a bound on $|f|$ over this ellipse. So, it looks like a standard $(n+1)$-point quadrature formula, but notice that M is not a bound on the derivative $f^{(2n+2)}$, but on f itself, which makes the inequality extremely valuable. The trick lies in the fact that knowing an analytic function over the complex plane (or at least an ellipse of it) tells how fast its Chebyshev approximants converge to it [Ber12, p. 94]. Implementing this method into CoqInterval would require some large formalization effort. First, a theory of analytic functions is needed (see also Section 3.1.3), as well as a theory on Gaussian quadrature. Second, instead of an

arithmetic on sets of real numbers (*i.e.*, interval arithmetic), an effective arithmetic on sets of complex numbers should be formalized (*i.e.*, ball arithmetic).

Improper integrals

Regarding improper integrals $\int_u^{+\infty} h$, the situation is not quite satisfactory, since the user has to explicit rewrite the integrand h as a product fg with f naively enclosable by interval arithmetic over $[u; +\infty)$ and g a function that is well known to be integrable at $+\infty$. It would be much more usable if the tactic was able to automatically find $\mathbf{c} \in \mathbb{I}$ and $g \in \mathbb{R} \to \mathbb{R}$ such that $h(t) \in \mathbf{c} \cdot g(t)$ for all $t \in [u; +\infty)$. In a sense, this is just a matter of defining an arithmetic over the pairs $\langle \mathbf{c}, g \rangle$, though this would not handle cancellations properly. The latter issue is not without relation to the automatic asymptotic expansion of exp-log functions [RSSvdH96].

Differential equations

While this chapter was focused on the topic of formally verified numerical approximations of definite integrals, but one might also wonder about indefinite integrals. The CoqInterval approach for enclosing a definite integral $\int_u^v f$ consists in computing the primitive of a polynomial approximation of the integrand f. So, by definition, the indefinite problem $x \mapsto \int_u^x f$ is also solved for $x \in [u; v]$, since we have a polynomial approximation of the primitive. But this does not work if the integration domain $[u; v]$ has been bisected. In that case, the indefinite problem is solved on the rightmost subdomain $[w; v]$ only. This might be sufficient for most applications, though.

The case of an *ordinary differential equations* $y'(x) = f(x, y(x))$ is slightly more complicated, since it is no longer just a matter of approximating f. To solve it, we can start from a crude polynomial enclosure $\mathbf{y}_0$ of the solution y that satisfies some initial condition $y_0 = y(x_0)$. Then, we iteratively compute better polynomial enclosures

$$\left(x \in \mathbf{x} \mapsto y_0 + \int_{x_0}^{x} f(t, \mathbf{y}_i(t))\, dt \right) \subseteq \mathbf{y}_{i+1},$$

until the error interval of $\mathbf{y}_i$ is small enough. Note that the use of Picard's operator reduces the problem to iteratively approximating an indefinite integral. Providing the inital enclosure $\mathbf{y}_0$, and making the error interval decrease when i increases, might prove challenging, though.

A slightly different approach has been used by Immler for the Isabelle/HOL system. Rather than using polynomial enclosures, only affine enclosures are used. Moreover, Picard's operator is mostly used to prove some properties of the solution. The actual approximation is performed using a two-sage Runge-Kutta method. This approach has made it possible to formally verify some parts of the chaos proof of Lorentz' contractor [Imm18, IT19].

Lastly, it should be noted that Picard's operator has already been used in the setting of constructive real analysis for Coq [MS13]. The use of a naive method for computing the improper integral (similar to Corollary 3.6), however, makes this method effective only for the simplest problems.

Multivariate integrals

Finally, one topic that has not yet been mentioned is the issue of multivariate integration. This should not come as a surprise, since, even outside the world of formal reasoning, there has not

been much work on devising reliable algorithms. Most of the works gravitate around Monte-Carlo methods and other probabilistic algorithms. In fact, even in the simplest case, *i.e.*, polynomial integrand and hypercube domain, there is no efficient approximation algorithm. On the contrary, NP problems can be reduced to this case [Fu12].

Chapter 4

Mathematical Libraries

In Chapter 3, three kinds of functions were formally verified: floating-point arithmetic operators, interval operators, and quadrature algorithms. The complexity of these functions did not originate from floating-point arithmetic, but from integer arithmetic or real arithmetic or real analysis. Indeed, most floating-point operations were rounded outwards in order to be used as interval bounds, which made their usage correct by construction. Difficulties related to floating-point arithmetic would have occurred only if we had tried to formally prove the completeness or the time complexity of these algorithms.

In this chapter, we change the focus toward mathematical libraries that are implemented using floating-point arithmetic rounded to nearest. This time, what matters is the final accuracy of a computed value with respect to an ideal value. Our running example will be an implementation of the exponential function for the *binary64* format. Figure 4.1 shows its C code, which is inspired by the implementation proposed by Cody and Waite [CW80, p. 65].

The code first eliminates input values whose exponential is either too small or too large to be representable in *binary64*. An argument reduction then brings the input x into a reduced domain by computing an integer k and a floating-point number $t \simeq x - k \cdot \log 2$ such that $t \in [-0.35; 0.35]$ holds. Thus, $\exp x \simeq 2^k \cdot \exp t$. The code approximates $\frac{1}{2} \exp t$ by performing a *binary64* evaluation of the rational function

$$f(t) = \frac{t \cdot p(t^2)}{q(t^2) - t \cdot p(t^2)} + 0.5,$$

where p and q are two polynomials with floating-point coefficients. Finally, the code returns a value that approximates $2^{k+1} \cdot f(t) \simeq \exp x$.

Figure 4.2 shows how the relative error between $\texttt{cw\_exp}(x)$ and $\exp x$ varies depending on the input x. More precisely, it shows about 750 points that are believed to be the worst local relative errors. They were obtained by testing 100 million floating-point numbers in the domain $[-708; 708]$. While 100 million points might seem like a huge sample, that is only 10^{-11} of all the *binary64* numbers in that domain. This explains the need to formally verify mathematical libraries for formats larger than *binary32*.

When verifying a function such as `cw_exp`, numerous questions arise. We will tackle the following ones in this chapter:

1. How far is $f(t)$ from $\frac{1}{2} \exp t$ for $t \in [-0.35; 0.35]$? As shown in Section 4.1, the CoqInterval library can formally verify a tight bound on the relative error.

```
double cw_exp(double x)
{
  double Log2h = 0xb.17217f7d1c00p-4;
  double Log2l = 0xf.79abc9e3b398p-48;
  double InvLog2 = 0x1.71547652b82fep0;
  double p1 = 0x1.c70e46fb3f692p-8;
  double p2 = 0x1.152a46f58dc1cp-16;
  double q1 = 0xe.38c738a128d98p-8;
  double q2 = 0x2.07f32dfbc7012p-12;

  // exceptional cases
  if (x < -746.) return 0.;
  if (x > 710.) return INFINITY;

  // argument reduction
  double k = nearbyint(x * InvLog2);
  double t = x - k * Log2h - k * Log2l;

  // rational evaluation
  double t2 = t * t;
  double p = 0.25 + t2 * (p1 + t2 * p2);
  double q = 0.5  + t2 * (q1 + t2 * q2);
  double f = t * (p / (q - t * p)) + 0.5;

  // result reconstruction
  return ldexp(f, (int)k + 1);
}
```

Figure 4.1 – A somehow accurate implementation of exp for *binary64*.

2. The evaluation of f is performed using floating-point operations, so how much inaccuracy is introduced there? Section 4.2 presents the Gappa tool and explains how the tool produces formal certificates on round-off error bounds.

3. Is t a sufficiently accurate reduced argument? The algorithm proposed by Cody and Waite is actually quite subtle, so the user might have to guide the tool in order to get a formal proof (Section 4.2.3).

One might also wonder whether the C compiler will mess with the code, making the whole verification effort pointless. Section 4.3 shows how the CompCert C compiler offers a usable semantics of floating-point arithmetic to C programs. In particular, the section details how Flocq's formalization presented in Section 3.2 was extended to cover the IEEE-754 formats, including exceptional cases such as infinities and NaNs.

The sequence "argument reduction, evaluation, result reconstruction" is quite specific and one might encounter other kinds of floating-point functions. Section 4.4 presents some

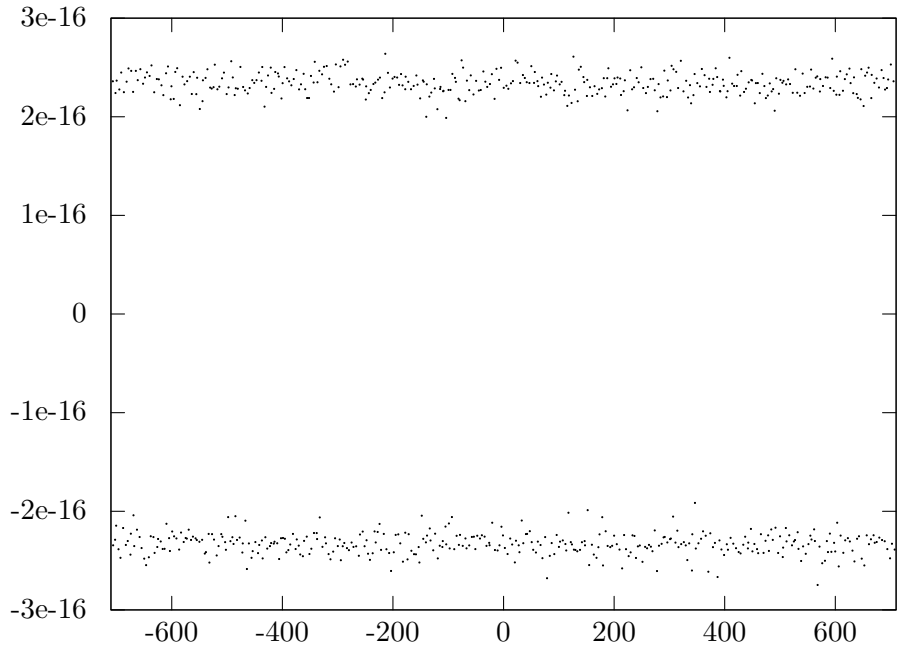


Figure 4.2 – Relative error between `cw_exp`(x) and $\exp x$.

examples of predicates that can be found in computational geometry and how their formal verification can be tackled.

Finally, since a tool such as Gappa is entirely dedicated to floating-point computations, the user might have to do some preprocessing work to expunge the algorithm from other considerations, *e.g.*, arrays, pointers. So, it would be interesting to see whether more generic tools, *e.g.*, SMT solvers, could be used instead. Section 4.5 explains how Alt-Ergo was modified to support floating-point arithmetic.

4.1 Method error

Section 4.2.3 will show how we can prove that the argument reduction of Figure 4.1 produces a value t such that $|t| \leq 355 \cdot 2^{-10} \simeq 0.35$. For now, we assume it. Our objective is to bound the relative error between $2f(t)$ and $\exp t$. Figure 4.3 shows how this error varies depending on the input t. As mentioned earlier, the rational function f is

$$f(t) = \frac{t \cdot p(t^2)}{q(t^2) - t \cdot p(t^2)} + 0.5,$$

with p and q two degree-2 polynomials with *binary64* coefficients. These coefficients, translated from C to Coq, are as follows, with `(pow2 k)` denoting 2^k.

```
Definition p0 := 1 * pow2 (-2).
Definition p1 := 4002712888408905 * pow2 (-59).
Definition p2 := 1218985200072455 * pow2 (-66).
Definition q0 := 1 * pow2 (-1).
```

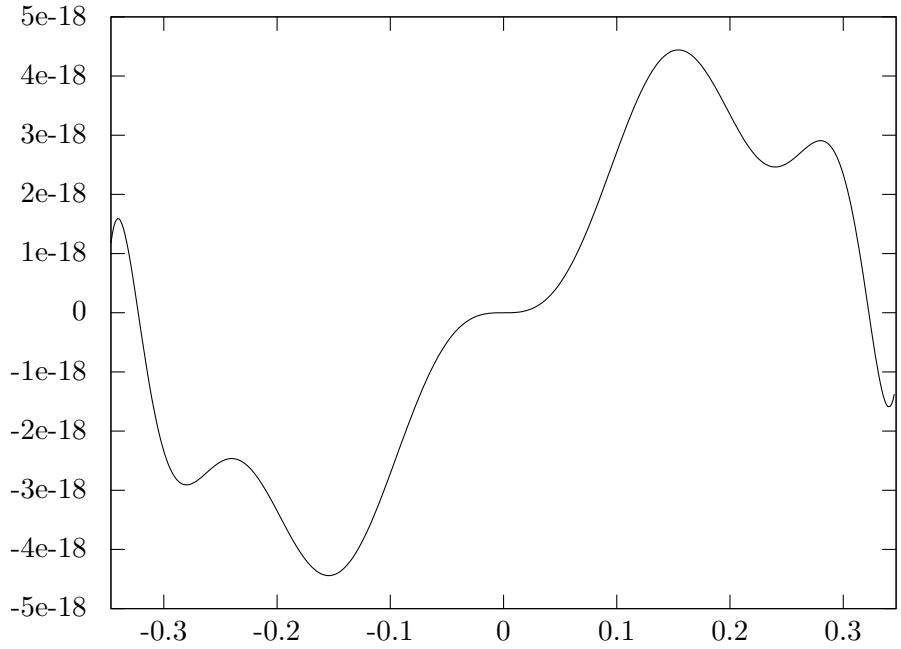


Figure 4.3 – Relative error between $2f(t)$ and $\exp t$.

```
Lemma method_error :
  forall t : R,
  let t2 := t * t in
  let p := p0 + t2 * (p1 + t2 * p2) in
  let q := q0 + t2 * (q1 + t2 * q2) in
  let f := (t * p) / (q - t * p) + 1/2 in
  Rabs t <= 355 / 1024 ->
  Rabs ((2*f - exp t) / exp t) <= 23 * pow2 (-62).
Proof.
intros t t2 p q f Ht.
unfold f, q, p, t2, p0, p1, p2, q0, q1, q2.
interval with (i_bisect_taylor t 9, i_prec 70).
Qed.
```

Figure 4.4 – Coq script proving that the relative method error is less than $23 \cdot 2^{-62}$.

```
Definition q1 := 8006155947364787 * pow2 (-57).
Definition q2 := 4573527866750985 * pow2 (-63).
```

Section 3.4 has shown how to use the `interval` tactic to automatically verify inequalities. That is precisely what we want to do, since we want to prove a bound on the relative error. Figure 4.4 shows both the Coq statement and its 3-line proof. This proof script tells the `interval` tactic to perform a bisection with respect to t. So that the bisection does not end up splitting the input interval into insanely many subintervals, the script also tells the tactic to use polynomial approximations rather than naive interval arithmetic. At least degree 5 is needed, and choosing degree 9 reduces the verification time to a few seconds.

Finally, the script tells the tactic to perform interval computations using a 70-bit precision floating-point arithmetic. Indeed, CoqInterval's default precision of 30 bits is not sufficient. In fact, since we are trying to prove a bound much smaller than 2^{-53}, even *binary64* numbers would cause a catastrophic cancellation. As an aside, this also means that *binary64* computations, and thus most plotting tools, would fail to properly produce Figure 4.3. This explains why I had to produce this plot with the Sollya tool [CJL10].

That is it. This is the shortest section of this document, as we need nothing more than what was already presented in Section 3.4, to perform the proof automatically.

4.2 Round-off error

Most proofs on floating-point algorithms involve some repetitive, tedious, and error-prone tasks, *e.g.*, verifying that no overflows occur. Another common task is a forward error analysis that proves that the distance between the computed result and the ideal result is bounded by a specified constant.

The Gappa[1] tool, which I have started developing during my PhD thesis under the supervision of Marc Daumas, is meant to automate part of this task [DM04, DM10]. Given a high-level description of a binary floating-point (or fixed-point) algorithm, it tries to prove or exhibit some properties of this algorithm using interval arithmetic and forward error analysis. When successful, the tool also generates a formal proof that can be verified by a formal system such as Coq. With the help of Sylvie Boldo and Jean-Christophe Filliâtre, I extended this mechanism so that the tool can directly be used as a Coq tactic to automatically discharge some arithmetic goals [BFM09].

In particular, we want to use this `gappa` tactic to automate most of the proof the Coq statement given in Figure 4.5. All the functions `add`, `sub`, `mul`, `div` denote *binary64* floating-point operators with rounding to nearest, tie breaking to even. The theorem states that, for any $x \in [-746; 710]$ representable as a *binary64* number, the relative error between `cw_exp`(x) and $\exp x$ is less than 2^{-51}. Note that the operators used in the Coq version ignore overflow. (The operations cannot overflow and that is easy to prove.) Moreover, the very last multiplication is not rounded, so it does not match the behavior of `ldexp` in case of underflow. In fact, for values of x so small that `cw_exp`(x) is in the subnormal range, the actual relative error can grow as large as 1. Nonetheless, the theorem implies that the absolute error is bounded in the subnormal case.

Section 4.2.1 gives an overview of the way Gappa looks for proofs. Section 4.2.2 focuses on the methods used for bounding round-off errors. They are sufficient to automate the

[1] `http://gappa.gforge.inria.fr/`

```
Definition cw_exp (x : R) :=
  let k := nearbyint (mul x InvLog2) in
  let t := sub (sub x (mul k Log2h)) (mul k Log2l) in
  let t2 := mul t t in
  let p := add p0 (mul t2 (add p1 (mul t2 p2))) in
  let q := add q0 (mul t2 (add q1 (mul t2 q2))) in
  pow2 (Zfloor k + 1) * (add (div (mul t p) (sub q (mul t p))) (1/2)).

Theorem exp_correct :
  forall x : R,
  generic_format radix2 (FLT_exp (-1074) 53) x ->
  -746 <= x <= 710 ->
  Rabs ((cw_exp x - exp x) / exp x) <= 1 * pow2 (-51).
```

Figure 4.5 – Correctness statement for the code of Figure 4.1

proofs related to the floating-point evaluation of the rational function that approximates exp. Unfortunately, for the argument reduction, Gappa is unable to complete the proofs. So, Section 4.2.3 shows how the user can help the tool by proving a few equalities.

4.2.1 Gappa's engine

While Gappa is meant to help verify floating-point algorithms, it manipulates expressions on real numbers only and proves properties of these expressions. Floating-point arithmetic is expressed using the functional rounding operators presented in Section 2.1. As a consequence, infinities, NaNs, and signed zeroes are not first-class citizens in this approach. Gappa is unable to propagate them through computations, but it is nonetheless able to prove they do not occur during computations.

Gappa takes as input a logical proposition about enclosures (for the most part), and it tries to generate a formal certificate that this proposition holds, whatever real numbers are substituted to the free variables. It does so by negating the proposition and then generating new facts using a database of theorems until a contradiction is found.

In this section, we will temporarily forget about Coq scripts. Propositions will be stated using Gappa's syntax, as it makes it easier to explain how the tool works.

Handling theorems

The database of theorems is inspired by interval arithmetic and can be written as definite Horn clauses. (As usual for such clauses, the consequent is written on the left.) For example, the rule for addition is as follows:

$$u + v \in \mathbf{w} \quad \Leftarrow \quad u \in \mathbf{u} \wedge v \in \mathbf{v} \wedge \mathbf{u} + \mathbf{v} \subseteq \mathbf{w}.$$

The validity of these rules has been formally verified in Coq and the generated certificate just enumerates all the rules needed to reach a contradiction. For example, assume that

Coq knows both facts $u \in [1;2]$ and $v \in [3;4]$ (either by hypothesis or because they have already been checked). Assume also that the certificate contains the fact $u + v \in [0;10]$ with the indication that the above theorem should be used to verify it. Coq proves that $[1;2] + [3;4] \subseteq [0;10]$ holds by computational reflection, from which it deduces $u + v \in [0;10]$.

In this setting, a contradiction is reached when an enclosure $e \in \emptyset$ is obtained. Using only the clauses performing interval forward propagation for the arithmetic operators, one cannot reach such a contradiction, since intervals only grow. So, Gappa can also make use of some set operations when it knows two enclosures of a single expression:

$$\begin{aligned} x \in \mathbf{x} &\Leftarrow x \in \mathbf{x_1} \wedge x \in \mathbf{x_2} \wedge \mathbf{x_1} \cap \mathbf{x_2} \subseteq \mathbf{x}, \\ x \in \mathbf{x} &\Leftarrow x \in \mathbf{x_1} \wedge x \notin \mathbf{x_2} \wedge \mathbf{x_1} \setminus \mathbf{x_2} \subseteq \mathbf{x}. \end{aligned}$$

Note that negated enclosures such as $x \notin \mathbf{x_2}$ are facts from the original proposition and are never produced by any of the theorems known by Gappa.

There are also some clauses able to eliminate disjunctions that are present in the original proposition. If we denote by P a conjunction/disjunction of literals containing a hole, and by $P[\ell]$ the same proposition with the hole filled by the literal ℓ, we get the following rules:

$$\begin{aligned} P[\top] &\Leftarrow x \in \mathbf{x_1} \wedge P[x \in \mathbf{x_2}] \wedge \mathbf{x_1} \subseteq \mathbf{x_2}, \\ P[\bot] &\Leftarrow x \in \mathbf{x_1} \wedge P[x \in \mathbf{x_2}] \wedge \mathbf{x_1} \cap \mathbf{x_2} = \emptyset, \\ P[\bot] &\Leftarrow x \in \mathbf{x_1} \wedge P[x \notin \mathbf{x_2}] \wedge \mathbf{x_1} \subseteq \mathbf{x_2}, \\ P[\top] &\Leftarrow x \in \mathbf{x_1} \wedge P[x \notin \mathbf{x_2}] \wedge \mathbf{x_1} \cap \mathbf{x_2} = \emptyset. \end{aligned}$$

Note that $P[\top]$ and $P[\bot]$ are meant to be simplified according to the rules of propositional logic, hopefully to a conjunction of literals. For example, consider a fact $x \in [0;4] \vee y \in [1;2]$ coming from the original proposition, and another fact $x \in [5;6]$ coming from the application of a theorem. Then, the second clause, with $P = (\bullet \vee y \in [1;2])$, produces $\bot \vee y \in [1;2]$, which is immediately simplified to the usable fact $y \in [1;2]$.

Proof search

Let us consider the Horn clause for addition again:

$$u + v \in \mathbf{w} \quad \Leftarrow \quad u \in \mathbf{u} \wedge v \in \mathbf{v} \wedge \mathbf{u} + \mathbf{v} \subseteq \mathbf{w}.$$

The relation $\mathbf{u} + \mathbf{v} \subseteq \mathbf{w}$ on the right-hand side is implemented by interval arithmetic. Since $\mathbf{u}$ and $\mathbf{v}$ appear in positive literals, this part of the clause is well suited for a *bottom-up* search (*à la* Datalog). To illustrate such a search, let us consider the proposition

$$x \in [0;1] \wedge y \in [2;3] \wedge (y + x) + ((x + x) + y) \notin [4;9] \Rightarrow \bot.$$

From the two enclosures $x \in [0;1]$ and $y \in [2;3]$, a bottom-up search deduces four new enclosures $x + x \in [0;2]$, $x + y \in [2;4]$, $y + x \in [2;4]$, and $y + y \in [4;6]$. Then, it deduces $x + (x + x) \in [0;3]$, $x + (x + y) \in [2;5]$, and 30 other new enclosures. And so on. Assuming the objective is to prove $x \in [0;1] \wedge y \in [2;3] \Rightarrow (y + x) + ((x + x) + y) \in [4;9]$, the bottom-up search eventually succeeds, but after producing thousands of useless enclosures, depending on the order clauses are instantiated.

If the procedure implements a *top-down* search instead (*à la* Prolog), it can focus on producing only the important facts. It starts from the negated enclosure and tries to contradict

it. To do so, it first instantiates the Horn clause for addition using $u_1 = y+x$, $v_1 = (x+x)+y$, and $\mathbf{w}_1 = [4; 9]$. But there is no clue on how to instantiate either $\mathbf{u}_1$ or $\mathbf{v}_1$, so the search gets stuck. Let us suppose it makes a guess, *e.g.*, $\mathbf{u}_1 = [1; 4]$ and $\mathbf{v}_1 = [3; 5]$, so as to progress further. It can now perform another instantiation of the Horn clause, this time using $u_2 = y$, $v_2 = x$, $\mathbf{u}_2 = [2; 3]$, $\mathbf{v}_2 = [0; 1]$, and $\mathbf{w}_2 = [1; 4]$. This part of the derivation tree is now closed, since $x \in [0; 1]$ and $y \in [2; 3]$ are actual hypotheses. But the procedure also needs to find a derivation for $(x + x) + y = v_1 \in \mathbf{v}_1 = [3; 5]$, which is impossible, whatever guesses it makes.

To summarize, a bottom-up search can succeed efficiently if an oracle tells it how to instantiate real variables, while a top-down search can succeed only if an oracle tells it how to instantiate interval variables. Gappa mixes both approaches to circumvent the lack of such oracles. It first performs a top-down search, trying to answer queries of the shape $e \in \bullet$ for any expression e that appears in the original proposition. During this top-down search, Gappa completely ignores any part of the clauses involving arithmetic. Once it has found all the possible ways of answering a query $e \in \bullet$, it creates a fresh set of clauses in which all the real variables have been substituted by the expressions encountered when answering the queries. From the facts $x \in \bullet$, $y \in \bullet$, and $(y + x) + ((x + x) + y) \notin \bullet$, the queries performed by a top-down search lead to the following Horn clauses:

$$\begin{aligned}
(y + x) + ((x + x) + y) \in \mathbf{w}_1 &\Leftarrow y + x \in \mathbf{u}_1 \wedge (x + x) + y \in \mathbf{v}_1 \wedge \mathbf{u}_1 + \mathbf{v}_1 \subseteq \mathbf{w}_1, \\
y + x \in \mathbf{w}_2 &\Leftarrow y \in \mathbf{u}_2 \wedge x \in \mathbf{v}_2 \wedge \mathbf{u}_2 + \mathbf{v}_2 \subseteq \mathbf{w}_2, \\
(x + x) + y \in \mathbf{w}_3 &\Leftarrow x + x \in \mathbf{u}_3 \wedge y \in \mathbf{v}_3 \wedge \mathbf{u}_3 + \mathbf{v}_3 \subseteq \mathbf{w}_3, \\
x + x \in \mathbf{w}_4 &\Leftarrow x \in \mathbf{u}_4 \wedge x \in \mathbf{v}_4 \wedge \mathbf{u}_4 + \mathbf{v}_4 \subseteq \mathbf{w}_4.
\end{aligned}$$

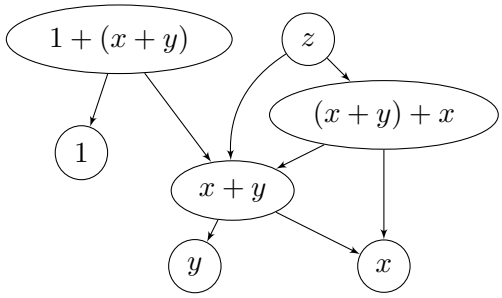
These new clauses only contain interval variables, so Gappa can now perform a bottom-up search on them. As traditional, whenever a new fact is produced by this bottom-up search, only clauses that mention it will be considered. For performance reasons, the clauses to consider are not put into a single FIFO, but into two. One has a higher priority than the other, which means that clauses are first retired from it. Whether a clause is put in the high-priority or low-priority FIFO depends on how long it has been since the last time the clause produced a useful fact. A new fact is useful if it is not trivially implied by an existing fact. For example, given a known fact $x \in [0; 4]$, a new fact $x \in [1; 4]$ would be useful, but a new fact $x \in [0; 5]$ would be useless. A clause is allowed to produce 50 useless facts (heuristically arbitrary value) before ending up with a low priority. Any low-priority clause that produces a useful fact gets promoted back to high priority. Note that, to avoid that a *slow convergence* effect starves the system, a new fact such as $x \in [0; 3.99]$ would also be considered useless, since it is not deemed a sufficient improvement with respect to $x \in [0; 4]$.

The example above might seem simple, but that is only because we have assumed that Gappa knows about a single Horn clause. But there is more than just interval addition. (Though here, it is the only useful clause.) If we run Gappa, we see that the bottom-up search actually receives 75 instantiated Horn clauses rather than just 4.

Bisection

Section 2.2.3 has presented bisection as the simplest way to reduce the dependency effect. To tell Gappa to use it, the user provides some property P and an expression e. Gappa will subdivide the domain of e until property P is proved on each subdomain. Gappa can then use P (or any antecedent of P) to progress further in its search for a contradiction.

If the user has not asked explicitly for a bisection, Gappa tries to find one that might help. For the property P, it simply chooses $\bot$. For e, it performs a depth-first traversal of the directed acyclic graph of all the expressions that appear in the original proposition in order to choose the expression that is encountered the most often, *i.e.*, the one with the most inbound edges. For example, let us suppose that there are two expressions: z defined as $(x + y) \cdot ((x + y) + x)$, and $1 + (x + y)$. They are represented internally by the following graph:



On that example, Gappa will choose to subdivide the domain of $x + y$, since it is the subexpression that most other subexpressions depend on. So, hopefully, it will be the one that reduces the dependency effect the most. (That said, x would presumably have been a better choice.)

Gappa also subdivides intervals when some enclosures are partly contradictory. For example, if it is able to prove $y \in [0; 10]$, yet the original proposition contains $y \notin [3; 7]$, then Gappa will subdivide the domain of y in three parts, so that it can consider the two subcases $y \in [0; 3]$ and $y \in [7; 10]$ separately.

Equalities

While bisection might help, the strength of Gappa resides in the use of equalities. Indeed, given an equality $x = y$, Gappa will look for an enclosure of y in order to refine an enclosure of x. (This implicitly means that equalities are oriented.) This is expressed by the following Horn clause:

$$x \in \mathbf{z} \quad \Leftarrow \quad y \in \mathbf{z} \wedge x = y.$$

This will be especially useful to propagate errors, as Section 4.2.2 will show. But it can also be used to implement backward propagators cheaply [BM17, §4.3.6]. For example, if we denote $\mathcal{U}[e]$ the fact that e appears in the original proposition, the following two clauses are available to Gappa:

$$\begin{aligned} u = (u \cdot v)/v \quad &\Leftarrow \quad v \neq 0 \wedge \mathcal{U}[u \cdot v], \\ v = (u \cdot v)/u \quad &\Leftarrow \quad u \neq 0 \wedge \mathcal{U}[u \cdot v]. \end{aligned}$$

Thus, not only can Gappa compute an enclosure of $u \cdot v$ from the enclosures of u and v, but it can also compute an enclosure of u (resp. v) from the enclosures of $u \cdot v$ and v (resp. u).

Up to now, only one predicate has been presented in the context of Gappa: the enclosure $x \in \mathbf{x}$. This example shows two other predicates: $x = y$ and $u \neq 0$. This last predicate could certainly be expressed as $u \in \mathbf{u} \wedge 0 \notin \mathbf{u}$. But it appears so often when dealing with divisions or relative errors that it has become a primitive predicate of Gappa [BM17, §4.3.3]. This

allows it to appear on the left-hand side of a clause, as enclosures do. Here are a few clauses about it:

$$\begin{aligned} u \neq 0 &\Leftarrow |u| \in \mathbf{u} \wedge 0 \notin \mathbf{u}, \\ u \cdot v \neq 0 &\Leftarrow u \neq 0 \wedge v \neq 0. \end{aligned}$$

Proof simplification

To conclude about Gappa's engine, a few words should be said about the generator of formal proofs. As mentioned above, the generated certificate contains a list of clause instances that the formal system, *e.g.*, Coq, has to apply to reach a contradiction. In practice, verifying the certificate is much slower than finding a contradiction, so Gappa does not stop after finding a contradiction. Instead, it searches for a shorter list of instances that lead to a contradiction.

For example, let us assume that the original certificate first obtains $x \in [0; 5]$, then obtains $x \in [3; 7]$, and finally deduces a contradiction from the intersection $x \in [3; 5]$. But it might be that the contradiction could also have been deduced from $x \in [3; 7]$. In that case, all the clauses that lead to $x \in [0; 5]$ are useless and can be removed from the final certificate. To find a shorter certificate, Gappa starts from the end of the certificate (the contradiction) and enlarges the intervals of the inputs of the corresponding clause as much as possible without invalidating the proof. Then, it looks whether the clauses leading to these enlarged inputs are actually needed or if they can be skipped. Gappa repeats this process for all the clauses in reverse order until it reaches the facts from the original proposition.

Note that Gappa enlarges the intervals by reducing the precision of the bounds. Thus, even if none of the clauses are found to be redundant, they might now be much faster to verify. Indeed, by default, Gappa works with a precision of 60 bits, but most of the proofs only care about the magnitude of the bounds. For example, suppose that the proof needs to enclose the absolute error $\square(\sqrt{x}) - \sqrt{x}$ for $x \in [9; 13]$. Verifying an enclosure of $\sqrt{x}$ computed with an accuracy of 60 bits is useless, since knowing only $\sqrt{x} \in [2; 4]$ leads to the exact same enclosure of the absolute round-off error.

Finally, bisection proofs can also be simplified in some cases. To do so, once Gappa has found a way to prove a property P by subdividing the domain of some expression e, it looks whether the number of subintervals can be reduced. It does so by merging contiguous subintervals as long as it is able to prove that P still holds.

4.2.2 Forward error analysis

Equalities are the mechanism by which Gappa performs backward propagation of enclosures, but more importantly, they permit to perform forward error analysis. Let us consider an example. We have four numbers u, v, $\tilde{u}$, and $\tilde{v}$, that satisfy the following properties:

$$u \in [1; 10] \wedge \tilde{u} \in [1; 10] \wedge v \in [5; 20] \wedge \tilde{v} \in [5; 20] \wedge \left|\frac{\tilde{u} - u}{u}\right| \leq 10^{-2} \wedge \left|\frac{\tilde{v} - v}{v}\right| \leq 10^{-3}.$$

We wish to bound the relative error between the two exact products $\tilde{u} \cdot \tilde{v}$ and $u \cdot v$. Naive interval arithmetic gives the following enclosure:

$$\frac{\tilde{u} \cdot \tilde{v} - u \cdot v}{u \cdot v} \in \frac{[1; 10] \cdot [5; 20] - [1; 10] \cdot [5; 20]}{[1; 10] \cdot [5; 20]} = \frac{[-195; 195]}{[5; 200]} = [-39; 39].$$

Such large bounds on the relative error are pointless. The fact that only the bounds on the values have been taken into account, but not their closeness, hints to a solution. We have to make explicit the dependencies between $\tilde{u}$ and u and between $\tilde{v}$ and v. We can improve the tightness of the interval evaluation by first rewriting the relative error as follows:

$$\frac{\tilde{u}\cdot\tilde{v}-u\cdot v}{u\cdot v}=\frac{\tilde{u}-u}{u}+\frac{\tilde{v}-v}{v}+\frac{\tilde{u}-u}{u}\cdot\frac{\tilde{v}-v}{v}. \tag{4.1}$$

There are still some dependencies, since the same relative errors appear twice in the rewritten expression. These dependencies cause the lower bound to be a bit underestimated, but at least the final bounds now have the correct order of magnitude:

$$\begin{aligned}\frac{\tilde{u}\cdot\tilde{v}-u\cdot v}{u\cdot v} &\in [-10^{-2};10^{-2}]+[-10^{-3};10^{-3}]\\ &\qquad +[-10^{-2};10^{-2}]\cdot[-10^{-3};10^{-3}]\\ &\in [-1.101\cdot10^{-2};1.101\cdot10^{-2}].\end{aligned}$$

Notice that the bounds on u, v, $\tilde{u}$, and $\tilde{v}$ are not even used anymore during the interval computations above. The bounds on u and v are still useful though, since they serve to prove that the product $u\cdot v$ is nonzero.

Tightening bounds

In a Gappa script, the proposition to prove is written between curly brackets. Intervals can be replaced by interrogation marks, in which case, Gappa answers the best enclosures it can prove fast. If we ignore the (useless) enclosures of $\tilde{u}$ and $\tilde{v}$ and if we denote $\tilde{u}$ and $\tilde{v}$ by `ut` and `vt`, the proposition gets translated into the following script:

```
{ u in [1,10] /\ v in [5,20] ->
  |(ut - u) / u| <= 1e-2 ->
  |(vt - v) / v| <= 1e-3 ->
  (ut * vt - u * v) / (u * v) in ? }
```

Gappa answers that it has obtained a proof of the proposition, assuming that the interrogation mark in the script is replaced by $[-1.099\cdot10^{-2};1.101\cdot10^{-2}]$ (or any larger interval). To reduce the dependency effect and obtain this tight enclosure of the relative error between $\tilde{u}\cdot\tilde{v}$ and $u\cdot v$, Gappa has automatically applied the rewriting step of Equation (4.1).

Note that the bounds returned by Gappa are a bit tighter than what the manual computation above gave. As a matter of fact, these bounds are optimal. The tightened lower bound comes from the following observation. For $\varepsilon_1\geq-1$ and $\varepsilon_2\geq-1$, the expression $\varepsilon_1+\varepsilon_2+\varepsilon_1\cdot\varepsilon_2$ is increasing with respect to both ε_1 and ε_2, as can be seen by computing its partial derivatives. As a consequence, if we have the enclosures $\varepsilon_1\in[\underline{\varepsilon}_1;\overline{\varepsilon}_1]$ and $\varepsilon_2\in[\underline{\varepsilon}_2;\overline{\varepsilon}_2]$ with $\underline{\varepsilon}_1\geq-1$ and $\underline{\varepsilon}_2\geq-1$, we get

$$\varepsilon_1+\varepsilon_2+\varepsilon_1\cdot\varepsilon_2\in[\underline{\varepsilon}_1+\underline{\varepsilon}_2+\underline{\varepsilon}_1\cdot\underline{\varepsilon}_2;\overline{\varepsilon}_1+\overline{\varepsilon}_2+\overline{\varepsilon}_1\cdot\overline{\varepsilon}_2].$$

Since both lower bounds $\underline{\varepsilon}_1=-10^{-2}$ and $\underline{\varepsilon}_2=-10^{-3}$ are negative, their product is positive, which mechanically tightens the lower bound on the relative error of the product, compared to the bounds obtained by interval arithmetic. The same kind of trick can be applied to other manipulations of relative errors, *e.g.*, division or composition.

Another way to obtain the tightened bounds would have been to rewrite $\varepsilon_1 + \varepsilon_2 + \varepsilon_1 \cdot \varepsilon_2$ into $(1 + \varepsilon_1) \cdot (1 + \varepsilon_2) - 1$ before doing the interval evaluation. The main issue with such a rewriting is that it induces a large cancellation when computing the bounds. For example, if ε_1 and ε_2 are bounded by 2^{-53}, getting a final bound of about 2^{-52} would require an internal precision of about 110 bits (instead of just a few bits), thus making the generated Coq proof much longer to verify.

Relative errors

As mentioned above, we still need enclosures on u and v to prove that they are nonzero and thus to talk about relative errors. (Or we could have directly written `(u <> 0 /\ v <> 0)` as a hypothesis.) But that is mostly due to the representation of relative errors as quotients. Thus, in addition to the predicates for enclosure and nonzeroness, Gappa also supports a relation that expresses bounded relative errors:

$$\mathrm{REL}(\tilde{u}, u, \mathbf{e_u}) \stackrel{\text{def}}{=} \exists \varepsilon_u \in \mathbf{e_u},\ \tilde{u} = u \cdot (1 + \varepsilon_u).$$

For the product, Gappa uses the following clause, which does not require the values u and v to be nonzero:

$$\mathrm{REL}(\tilde{u} \cdot \tilde{v}, u \cdot v, \mathbf{e}) \quad \Leftarrow \quad \mathrm{REL}(\tilde{u}, u, \mathbf{e_u}) \wedge \mathrm{REL}(\tilde{v}, v, \mathbf{e_v}) \wedge \mathbf{e_u} + \mathbf{e_v} + \ldots \subseteq \mathbf{e}.$$

In the syntax of Gappa, $\mathrm{REL}(\tilde{u}, u, \mathbf{e_u})$ is denoted `(ut -/ u in eu)`. The syntactic sugar `(|ut -/ u| <= e)` denotes the relation $\mathrm{REL}(\tilde{u}, u, [-e; e])$. We can now rewrite the original Gappa script as follows.

```
{ |ut -/ u| <= 1e-2 ->
  |vt -/ v| <= 1e-3 ->
  ut * vt -/ u * v in ? }
```

For division, the clause is similar to multiplication, except that the relative errors are subtracted. Gappa also supports addition and subtraction, but the formulas are much more complicated. In addition to the enclosures of $(\tilde{u}/u - 1)$ and $(\tilde{v}/v - 1)$, Gappa also needs an enclosure of the quotient $u/(u + v)$ to quantify the degree of cancellation [BM17, §4.3.2]. Indeed, it relies on the following equality to compute the relative error of the sum:

$$\frac{(\tilde{u} + \tilde{v}) - (u + v)}{u + v} = \frac{u}{u + v} \cdot \frac{\tilde{u} - u}{u} + \left(1 - \frac{u}{u + v}\right) \cdot \frac{\tilde{v} - v}{v}.$$

This rewriting of the relative error works fine as long as Gappa is able to compute a tight enclosure of $u/(u + v)$. To increase its chances of success, the tool tries not only to enclose $u/(u+v)$, but it also considers several variants of it and takes the intersection of their enclosures (if needed):

$$\frac{u}{u + v} = \left(1 + \frac{v}{u}\right)^{-1} = 1 - \frac{v}{u + v} = 1 - \left(1 + \frac{u}{v}\right)^{-1}.$$

Error composition

The last important point about error analysis is the composition of errors. We will be looking at absolute errors for simplicity, but Gappa applies similar methods to relative errors. Let us

suppose that we need to bound a round-off error $\square(u) - v$, where u is an approximation of v. A usually fruitful way is to use the following equality, as it separates the rounding error from the method error:

$$\square(u) - v = (\square(u) - u) + (u - v).$$

But the approach is more general than that. If we are trying to bound the absolute error between an approximate value x and an ideal value z, we can try to decompose the error using a value y that either approximates z or is approximated by x:

$$x - z = (x - y) + (y - z).$$

A variant of this problem is the computation of an enclosure of x (resp. y) knowing that x approximates y. Then the following equalities might help:

$$x = y + (x - y) \quad \wedge \quad y = x - (x - y).$$

In all of these cases, the issue lies in enumerating all the interesting expressions for y. With $\square(u)$, that was easy, we just had to remove the head rounding operator to get a value approximated by $\square(u)$. In the general case, however, y might not be structurally related to either x or z.

Thus, Gappa keeps track of which expressions are approximated by which other expressions. This relation between expressions only serves to instantiate clauses, so getting it wrong might just prevent Gappa from finding a proof, due to too many or too few instantiated clauses. Let us denote $x \sim y$ the fact that x supposedly approximates y. The following clause is available to Gappa:

$$x \in \mathbf{x} \quad \Leftarrow \quad y + (x - y) \in \mathbf{x} \wedge x \sim y.$$

There are three main sources of relations $x \sim y$. First, as already mentioned, Gappa assumes $x \sim y$ when the head symbol of x is of the shape $\square(y)$. These relations also come from any enclosure of $x - y$ or $(x - y)/y$ that appears in the original proposition, since Gappa assumes that it represents an absolute/relative error. Equalities $x = y$ are the third source.

From these sources, Gappa computes the congruence closure of this relation. Suppose that both $E[\vec{u}]$ and $E[\vec{v}]$ occur in the original proposition, for some expression E containing holes. If Gappa knows $\forall i,\ u_i \sim v_i$, then it assumes $E[\vec{u}] \sim E[\vec{v}]$. As a consequence, writing trivial hypotheses is a way to force Gappa to consider additional relations of approximation.

Back to the example

Now that we have seen how Gappa performs forward error analysis, let us have a better look at the example. While the proof was entirely done within Coq, we still use Gappa as a standalone tool here, for the sake of clarity.

A Gappa script contains three parts. Up to now, only the middle part, delimited by curly brackets, has been used. This middle part describes the formula one wants to verify. It is a logical proposition whose atoms are enclosures of expressions (or some of the other predicates supported by Gappa). The first part of the script makes it possible to define some notations for expressions and rounding operators, so as to make the proposition more readable. As for the third part of the script, it is used to pass hints to the tool, *e.g.*, a bisection directive. Note that Gappa is not a tool for checking satisfiability, so any free variable is implicitly understood as being universally quantified.

The full script will be given at the end, on Figure 4.6. We start this script by defining the polynomial coefficients, as we will need to mention them several times.

```
p1 = 0x1.c70e46fb3f692p-8;
p2 = 0x1.152a46f58dc1cp-16;
q1 = 0xe.38c738a128d98p-8;
q2 = 0x2.07f32dfbc7012p-12;
```

We also give a name to the result of the infinitely precise evaluation of the rational function, as well as of the intermediate expressions. To distinguish them from the computed values, their names are capitalized.

```
T2 = t * t;
P = 0.25 + T2 * (p1 + T2 * p2);
Q = 0.5 + T2 * (q1 + T2 * q2);
F = (t * P) / (Q - t * P) + 0.5;
```

We now give a name to rounded versions of these expressions. A rounding operator for an FLT format is denoted `float`, followed by the precision, the minimal exponent, and the rounding mode. For *binary64* with rounding to nearest, tie breaking to even, this gives `float<53,-1074,ne>`. This can be simplified as `float<ieee_64,ne>`, but this is still a bit long to type, so we give it a name.

```
@rnd = float<ieee_64,ne>;
```

We can now use it to write rounded expressions, such as `rnd(t * t)`. Applying `rnd` to each subexpression is tedious, though, so Gappa makes it possible to specify a rounding operator on the left-hand side of a definition. In that case, it is applied to any arithmetic operator on the right-hand side. We can now type the remaining definitions:

```
t2 rnd= t * t;
p rnd= 0.25 + t2 * (p1 + t2 * p2);
q rnd= 0.5 + t2 * (q1 + t2 * q2);
f rnd= (t * p) / (q - t * p) + 0.5;
```

We now have enough notations to express the proposition we want to prove in a concise way. So, let us type the middle part of the script. We want to verify a bound on the relative error between f and F given an enclosure of t. Rather than giving the bound to prove, let us start by asking Gappa what it can prove. (`355b-10` denotes $355 \cdot 2^{-10}$.)

```
{ t in [-355b-10,355b-10] -> f -/ F in ? }
```

Unfortunately, Gappa answers that it cannot prove any bound. To understand what went wrong, we can pass the `-Munconstrained` option to the tool. This option allows Gappa to assume that neither division by zero nor underflow can occur. The downside is that Gappa can no longer generate a correct proof, as it would contain holes in a dozen places where the tool used the assumption. In this degraded mode, Gappa answers that the relative error is less than 2^{-51}, which is the kind of result we expect for this floating-point code.

We can now ask various questions to Gappa to try to pinpoint where it failed to handle a division or an underflow. For example, due to the dependency effect, it might have been unable to prove that $\circ[q - t \cdot p]$ does not come close to zero. Or it might have failed to bound the corresponding round-off error.

```
{ t in [-355b-10,355b-10] ->
    q - t * p in ? /\
    q - rnd(t * p) -/ Q - t * P in ? }
```

Gappa gives sensible answers, so the issue does not come from here. After a few attempts, we find that it comes from the dividend of the rational function. Indeed, if $t \cdot p$ underflows, its relative error is unbounded, and so is the relative error of the division. Gappa is right, but what it fails to understand is that, if the dividend underflows, then the result of the division is negligible with respect to 0.5, so the last addition absorbs any kind of error.

Thus, we just have to force Gappa to consider separately whether $t \cdot p$ is small or not. There are several ways to do so. For example, we can tell the tool how to split the domain of t. This is the purpose of the last part of the script. We tell Gappa to consider three subintervals, $t \in (-\infty; -2^{-60}] \cup [-2^{-60}; 2^{-60}] \cup [2^{-60}; +\infty)$, by adding the following hint.

```
$ t in (-1b-60,1b-60);
```

Gappa is then able to prove that the relative error between f and F is less than $2^{-51.5}$, which is a tight bound. We are not yet done, though, since this bound does not tell us what the accuracy of the whole function is. The relative error we are actually interested in is the one between $2f$ and $\exp(x - k\log 2)$. Fortunately, Gappa can still be of help.

To do so, we first introduce two new free variables: `exp_t` represents $\exp t$, and `exp` represents $\exp(x - k\log 2)$. We then tell Gappa about the relative error between $2F$ and $\exp t$ (see Section 4.1). We also give a bound on the relative error between $\exp t$ and $\exp(x - k\log 2)$, which can be derived from the absolute error between t and $x - k\log 2$ (see Section 4.2.3).

Given the complete script of Figure 4.6, Gappa instantly answers that the relative error is less than $2^{-51.3} \simeq 3.5 \cdot 10^{-16}$. The generated Coq proof is about 3,000-line long. Instead of letting Gappa analyze the code, we can also write a specific consequent with a tighter bound, *e.g.*, `|2*f -/ exp| <= 3.3e-16`. The tool still finishes instantly. In addition to the split on t given by the hint, the tool also decided to split the domain of t^2 (`T2`) into two parts in order to refine the bound on the relative error. The generated proof is now about 4,000-line long.

4.2.3 Argument reduction

The implementation of the exponential starts by approximating $x - k\log 2$. Computing this reduced argument as $\circ[x - k \cdot \circ(\log 2)]$ would be disastrous. Indeed, there is a large cancellation during the subtraction, which would cause the round-off error to explode. Using a fused multiply-add operator would avoid this issue, but the reduced argument would still be poor, as $\circ(\log 2)$ is just not accurate enough. For example, for $x = 700$, the absolute error between $\circ[x - k \cdot \circ(\log 2)]$ and $x - k\log 2$ is about 2^{-44}, from which a relative error of 2^{-51} on the final result is impossible to reach.

Cody and Waite's algorithm is one of the first attempts at performing an accurate argument reduction [CW80, p. 61]. Note that this was done at a time when FMAs did not exist. The relevant part of Figure 4.1 is reproduced below.

```
double InvLog2 = 0x1.71547652b82fep0;
double Log2h = 0xb.17217f7d1cp-4; // 42 bits out of 53
double Log2l = 0xf.79abc9e3b398p-48;
double k = nearbyint(x * InvLog2);
double t = (x - k * Log2h) - k * Log2l;
```

```
p1 = 0x1.c70e46fb3f692p-8;
p2 = 0x1.152a46f58dc1cp-16;
q1 = 0xe.38c738a128d98p-8;
q2 = 0x2.07f32dfbc7012p-12;

T2 = t * t;
P = 0.25 + T2 * (p1 + T2 * p2);
Q = 0.5 + T2 * (q1 + T2 * q2);
F = (t * P) / (Q - t * P) + 0.5;

@rnd = float<ieee_64,ne>;
t2 rnd= t * t;
p rnd= 0.25 + t2 * (p1 + t2 * p2);
q rnd= 0.5 + t2 * (q1 + t2 * q2);
f rnd= (t * p) / (q - t * p) + 0.5;

{ t in [-355b-10,355b-10] /\
  |2 * F -/ exp_t| <= 23b-62 /\
  |exp_t -/ exp| <= 33b-60
  ->
  2 * f -/ exp in ? }

$ t in (-1b-60,1b-60);
```

Figure 4.6 – Gappa script for the relative error of `cw_exp`.

The code first computes the integer k, which might not be equal to $\lfloor x/\log 2 \rceil$ due to numerical errors. Having an off-by-one value of k is not an issue, though, as long as $|x - k\log 2|$ is not much larger than $\frac{1}{2}\log 2$. To compute an approximation of $x - k\log 2$, the algorithm relies on a highly accurate approximation of $\log 2$: $|\texttt{Log2h} + \texttt{Log2l} - \log 2| \leq 2^{-102}$. Moreover, the `Log2h` constant is chosen so that its radix-2 significand ends with 11 zero bits. As a consequence, its floating-point product with the integer k is performed exactly, since $|k| \leq 2^{11}$. The floating-point subtraction to x is also exact, due to Sterbenz' theorem [Ste74]. Thus the only round-off errors come from the product `k * Log2l` and its subtraction.

Exact operations

Let us digress a bit to explain how Gappa handles exact computations. We have seen that Gappa supports a few predicates: $x \in \mathbf{x}$, $x \neq 0$, and $\mathrm{REL}(x, y, \mathbf{e})$, *i.e.*, $\exists \varepsilon \in \mathbf{e},\ x = y \cdot (1 + \varepsilon)$. There are two other predicates that are meant to account for the discreteness of floating-point

formats.[2] They relate an expression x and an integer k as follows:

$$\begin{aligned} \mathrm{FIX}\,(x,k) &\stackrel{\text{def}}{=} \exists m,e\in\mathbb{Z},\ x = m\cdot 2^e \wedge k \le e,\\ \mathrm{FLT}(x,k) &\stackrel{\text{def}}{=} \exists m,e\in\mathbb{Z},\ x = m\cdot 2^e \wedge |m| < 2^k. \end{aligned}$$

In other words, $\mathrm{FIX}(x,k)$ means that the real x can be represented by a binary fixed-point number with a least significant bit of weight 2^k. As for $\mathrm{FLT}(x,k)$, it means that x can be represented by a binary floating-point number with an integer significand of at most k bits. Note that, as with enclosures, only the expression x is meant to be symbolic; the value of the integer k has to be known by Gappa. If a property $\mathrm{FIX}(x,k)$ holds, any property $\mathrm{FIX}(x,\ell)$ with ℓ smaller than k also holds. For FLT, it is the opposite: if a property $\mathrm{FLT}(x,k)$ holds, any property $\mathrm{FLT}(x,\ell)$ with ℓ larger than k also holds.

The following lemmas are used by Gappa to deduce FIX and FLT properties on expressions involving arithmetic operators:

$$\begin{aligned} \mathrm{FIX}(u+v,m) &\Leftarrow \mathrm{FIX}\,(u,k)\wedge\mathrm{FIX}\,(v,\ell)\wedge m\le\min(k,\ell),\\ \mathrm{FIX}\,(u\cdot v,m) &\Leftarrow \mathrm{FIX}\,(u,k)\wedge\mathrm{FIX}\,(v,\ell)\wedge m\le k+\ell,\\ \mathrm{FLT}(u\cdot v,m) &\Leftarrow \mathrm{FLT}(u,k)\wedge\mathrm{FLT}(v,\ell)\wedge m\ge k+\ell. \end{aligned}$$

Let us consider Cody and Waite's argument reduction now. From the enclosure of x and the value of `InvLog2`, Gappa deduces $k\in[-2^{-11};2^{11}]$. Since k is a value rounded to the nearest integer, Gappa also deduces $\mathrm{FIX}(k,0)$. From both properties, it then deduces $\mathrm{FLT}(k,11)$. Since we have $\mathrm{FLT}(\texttt{Log2h},42)$ by definition, Gappa deduces $\mathrm{FLT}(k\cdot\texttt{Log2h},53)$. Finally, it is just a matter of instantiating the following clause to prove that the computations are exact:

$$\square(a)=a \quad\Leftarrow\quad \mathrm{FIX}(a,-1074)\wedge\mathrm{FLT}(a,53).$$

Formal statement

Now that we have seen how Gappa detects exact computations, we turn to the formal proof of the argument reduction. Our goal is to verify using Coq that the absolute error between t and $x-k\cdot\log 2$ is bounded by $(1+2^{-16})\cdot 2^{-55}$. We will also prove $|t|\le 355\cdot 2^{-10}$, as this is an assumption used in Section 4.1. The Coq statement is as follows, with `sub` and `mul` denoting floating-point subtraction and multiplication in the FLT format representing *binary64* numbers.

```
Lemma argument_reduction :
  forall x : R,
  generic_format (FLT_exp (-1074) 53) x ->
  -746 <= x <= 710 ->
  let k := nearbyint (mul x InvLog2) in
  let t := sub (sub x (mul k Log2h)) (mul k Log2l) in
  Rabs t <= 355 / 1024 /\
  Rabs (t - (x - k * ln 2)) <= 65537 * pow2 (-71).
```

[2] They should not to be confused with the FIX and FLT formats. While the meaning of FIX is quite close, Gappa's FLT predicate is closer to Flocq's FLX format. Gappa's naming predates Flocq's more sensible one.

We would like the whole formal proof to be automated. Unfortunately, Gappa is unable to prove the goal when x is close to 0. More precisely, given the set of lemmas at its disposal, the tool fails to notice that the floating-point subtraction $\circ[x - k \cdot \texttt{Log2h}]$ is actually exact. That is not much of an issue though. Indeed, t is equal to x in that case ($k = 0$), so the correctness of the argument reduction trivially follows.

Therefore, the very first step of the Coq proof is a case distinction depending on whether $|x| \leq T$ or $|x| \geq T$, for some threshold T. This threshold is chosen to be small enough so that $|x| \leq T$ implies $k = 0$ (trivial case), yet large enough for Gappa to prove that the subtraction is exact. After some trials and errors, we find that $T = 5/16$ is a suitable threshold. When $|x| \leq 5/16$, the most intricate part of the proof is to deduce $k = 0$, but we can use the `gappa` tactic to discharge this equality. So we are left to prove that the argument reduction also behaves properly when $x \in [-746; -5/16] \cup [5/16; 710]$.

User hints

Despite removing the $(-5/16; 5/16)$ domain, Gappa is still unable to automatically verify the bound on the absolute round-off error on the argument reduction. So, we have to provide some additional hypotheses to the tool to guide it. To find which ones are needed, let us guess which kind of reasoning the tool tries to apply and where it might fail to progress.

First of all, in order to verify $|t| \leq 355 \cdot 2^{-10}$ in the above Coq statement, Gappa must be able to compute some tight bounds on the following subexpression of t (ignoring the rounding operators, for clarity):

$$x - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h}.$$

This is a subtraction between values that are close to each other (especially when x is large), but Gappa cannot see it. Fortunately, it becomes obvious if one replaces the first occurrence of x with $x \cdot \texttt{InvLog2} \cdot \texttt{InvLog2}^{-1}$. The two subtracted subexpressions now have the same shape:

$$(x \cdot \texttt{InvLog2}) \cdot \texttt{InvLog2}^{-1} - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h},$$

so Gappa has no trouble giving some tight bounds on it. Therefore, the bound on t will be automatically verified as long as we assert $x = x \cdot \texttt{InvLog2} \cdot \texttt{InvLog2}^{-1}$. Note that this hint is not strictly needed, but it avoids a costly bisection from Gappa.

Let us now look at the rightmost side of the above Coq statement: $|t - (x - k \cdot \log 2)| \leq 65537 \cdot 2^{-71}$. This time, the tool has to compute some tight bounds on the following expression (again ignoring the rounding operators):

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - (x - k \cdot \log 2).$$

As before, the two subtracted subexpressions do not have the same shape, so Gappa will not be able to help us. We can exhibit this similarity by replacing $\log 2$ with $\texttt{Log2h} + \delta$. The expression then becomes

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - ((x - k \cdot \texttt{Log2h}) - k \cdot \delta).$$

Gappa has no trouble bounding such an expression, as long as we tell the tool how close `Log2l` is to $\delta = \log 2 - \texttt{Log2h}$. Indeed, since Gappa does not know anything about the logarithm, it does not have any way to actually compute an enclosure of either δ or $\texttt{Log2l} - \delta$. Fortunately, the `interval` tactic (see Section 3.4) makes it straightforward to prove $-2^{-102} \leq \texttt{Log2l} - \delta \leq 0$.

The resulting formal proof for argument reduction is about 20-line long, about as long as a detailed hand-written proof. Half of this Coq script deals with the case $|x| \leq 5/16$. For the case $|x| \geq 5/16$, the `gappa` tactic succeeds in discharging

$$|t| \leq 355 \cdot 2^{-10} \wedge |t - (x - k \cdot \log 2)| \leq 65537 \cdot 2^{-71},$$

once the following additional properties have been proved by the user:

- $x = x \cdot \texttt{InvLog2} \cdot \texttt{InvLog2}^{-1}$ using the standard Coq tactic `field` [GM05],
- $x - k \cdot \log 2 = x - k \cdot \texttt{Log2h} - k \cdot (\log 2 - \texttt{Log2h})$ using `field` (or rather `ring`),
- $\texttt{Log2l} - (\log 2 - \texttt{Log2h}) \in [-2^{-102}; 0]$ using `interval`.

4.2.4 Assessment

There have been various uses of Gappa as a standalone tool to prove the correctness of algorithms. Here are a few examples of mathematical functions partly verified using Gappa: the floating-point logarithm from CRlibm [dDLM06, dDLM11], a floating-point square root and some trigonometric functions for integer processors [JKMR11, JJL12], an implementation of integer division using floating-point arithmetic [BM17, §6.4]. More generally, it has been used as a component of code generators for mathematical functions [MR11, KL14]. It has also been used as backend for Frama-C/Why3 to verify some floating-point algorithms: the CPR algorithm of the ADS-B aircraft system [TMM+18], a numerical scheme for the wave equation [BCF+13] (see also Section 5.3.1).

Equalities

As explained, one of Gappa's strengths resides in the ability for the user to add equalities. This mechanism, however, is not as strong as one could expect. For example, Gappa might discover during its bottom-up phase that $x - y \in [0; 0]$ holds, *i.e.*, $x = y$. Unfortunately, the tool cannot make use of that knowledge to instantiate new clauses, as this happens only during the top-down phase, which has already been run. An obvious solution would be to alternate the top-down and bottom-up phases until no new equalities are discovered, but perhaps approaches from the logic programming world would help in better supporting equality [BMSU86].

Elementary functions

Most of the limitations of Gappa are a consequence, not of its architecture, but of its ability to produce proofs. Indeed, whichever deduction Gappa performs, Coq[3] has to be able to check it afterwards. This explains why Gappa has no support for elementary functions. Enclosures could have been computed using Boost.Interval and MPFR, but there would have been no way to formally verify these computations, in the early days of Gappa. Nowadays, the situation is quite different, since CoqInterval could well be used to verify these deductions. Due to the oracle-based model, the only difficulty would be for Gappa or Coq to guess the precision at which CoqInterval should perform its computations. We could even go further: use Taylor

[3] Or any other prover, as Gappa does not use any specific feature of Coq.

models inside Gappa and have CoqInterval verify them. This would considerably reduce the amount of work the user has to perform explicitly.

As an aside, note that Linderman developed a variant of Gappa, named Gappa++, which replaced all the interval computations with affine arithmetic [LHD+10]. It also supported exp and log. All of that work was done at the expense of formal proof generation, though.

Round-off error expansions

Such a use of Taylor models would only be meant to reduce the dependency effect when bounding the method error, as was done in Section 4.1. But people have also considered Taylor expansions to bound the round-off error [Lin76]. The idea is that each rounding operator produces some perturbation. By propagating expansions of these perturbations, we would then know the impact that each local rounding error has on the final round-off error. The larger coefficients would tell us which operation might cause the worst inaccuracy in the computed result. An application of this approach can be found in the Fluctuat tool [GP06]. Contrarily to Linnainmaa's approach, Fluctuat uses affine arithmetic, which means that the expansion is truncated at order 1.

Regarding affine arithmetic, Solovyev *et al* bring an interesting new take on the topic with the FPTaylor tool [SJRG15]. There are two major improvements with respect to Fluctuat. First, like Gappa, FPTaylor generates proof certificates, which can be checked using HOL Light. Second, the coefficients of the affine forms are not numerical but symbolical. Thus, the contribution of each local rounding error is bounded by an expression that depends on the inputs of the algorithm. This expression can then be fed to a global optimizer, in order to turn it into an actual numerical bound.[4] The Real2Float tool follows a similar approach, but it uses SDP solvers during global optimization, which might lead to certificates that are faster to check [MCD17].

A shortcoming of FPTaylor is that the floating-point model is rather naive, so there is no way for the tool to detect exact computations. So, Lee *et al* extended this approach with various rules such as Sterbenz' theorem to avoid creating spurious noise symbols [LSA18]. That way, they were able to automatically verify implementations of mathematical functions with sub-ulp accuracy. Again, formal proof generation was sacrificed in the process. In fact, it is doubtful whether this approach is amenable to verification using a formal system, since it took them about one year of CPU time to fully analyze a single function.[5] So, there is still a sweet spot to be found between Gappa and FPTaylor.

Loops

A shortcoming of both Gappa and FPTaylor is that they only handle straight-line programs. In particular, they cannot find invariants that could be used to verify program loops. In its most basic form, a loop invariant is an implication $\vec{y}_{\text{old}} \in \vec{\mathbf{y}} \Rightarrow \vec{y}_{\text{new}} \in \vec{\mathbf{y}}$. The peculiarity is that the intervals $\vec{\mathbf{y}}$ appear in both the antecedent and the consequent, which explains why the tools cannot be queried for them. That said, even if Gappa cannot be used to find invariants, it might at least be used to verify the ones provided by the user.

[4] On the last example of Section 4.2.2, FPTaylor computes the same bound as Gappa for the absolute round-off error between f and F. For the corresponding relative error, its bound is slightly better, $2.7 \cdot 10^{-16}$ instead of $2.9 \cdot 10^{-16}$, but FPTaylor does not generate any formal certificate for relative errors.

[5] Verifying an implementation of log took 461 hours of computations using 16 instances of Mathematica in order to analyze more than 4 million subintervals.

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

Table 4.1 – Results of the code of Figure 4.7 depending on the processor and compiler options.

The Fluctuat static analyzer does not suffer from this shortcoming. As a tool based on abstract interpretation [CC77], it is able to find loop invariants by widening the enclosures of $\vec{y}$ until they are preserved by the loop. In the worst case, the obtained invariant might be trivial and thus useless to verify the subsequent computations. But at least, Fluctuat can natively verify floating-point C programs. Not only does it handle loops, it can also deal with branching divergence, *i.e.*, tests that give different results depending on whether the values are computed with infinite precision or not.

While both Fluctuat and FPTaylor use affine arithmetic, it is not obvious how the later tool could be made to support loops. Indeed, its strength (affine forms with symbolic coefficients) is now its weakness. With numerical coefficients, sets represented by affine forms can be merged, widened, narrowed, and so on, which is needed to find invariants. With symbolic coefficients, there is no immediate implementation of these operations, even when using approaches from template abstract domains.

4.3 Machine code

A critical assumption was made in Section 4.2, that is, the semantics of the floating-point operations used by the algorithm can effectively be mapped to the semantics of IEEE-754 arithmetic operators. If not, then the whole verification effort was for naught. The good thing about IEEE 754 is that it is an international standard that has been around for almost thirty years. Moreover, mainstream chip manufacturers are willing to support it, so there are good chances that the processor on which the program will be run is actually compliant. So, the matter is more of whether the semantics of floating-point operations was preserved when going from the source code to the machine code. Section 4.3.1 explains what might go wrong and presents one possible solution: verified compilation. Section 4.3.2 then presents what was done in the context of the CompCert C compiler to support floating-point arithmetic.

4.3.1 C programs and floating-point arithmetic

The C code of Figure 4.7 is a small example in C illustrating how the compilation process might impact the code written by the user. The example implements the opposite of the Fast2Sum algorithm and applies it to the inputs $x = 1$ and $y = 2^{-53} + 2^{-78}$. Since $|x| \geq |y|$, the program is expected to output the opposite z of the rounding error of the floating-point sum of x and y in *binary64* with rounding to nearest, tie breaking to even. Unfortunately, as shown in Table 4.1, this very simple program compiled with GCC 8.1.0 gives three different answers on an x86 architecture depending on the instruction set and the chosen level of optimization. Of these three, only one corresponds to the expected output: `0x1.ffffffp-54`.

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

Figure 4.7 – Opposite of Fast2Sum of 1 and $2^{-53} + 2^{-78}$

The root cause for the discrepancies in the first three rows of the result table lies in the x86 architecture. Indeed, nowadays this architecture provides two floating-point units: one performs *binary32* and *binary64* computations, while the other one performs floating-point computations using 80-bit numbers (64 bits of precision). So, the result will depend on which floating-point unit and which format the compiler selects for performing each floating-point operation. The compiler may thus choose to round the infinitely precise result either to extended precision, or to double precision, or first to extended and then to double precision. Note that, in all cases, y is computed exactly, so $y = 2^{-53} + 2^{-78}$ and the compilation choices only impact the computation of z. With the `-O0` optimization level 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 level for the 64-bit instruction set, all the computations are performed with double precision. With level `-O1` and higher, the intermediate value $\circ[(x + y) - x]$ is precomputed 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 level `-Ofast`, the architecture no longer matters, since the program performs no floating-point computation; the executable code only outputs the constant 0. Indeed, this optimization level turns `-funsafe-math-optimizations` on, which allows the reorganization of floating-point operations. Hence, GCC dutifully reorganizes $\circ[((x + y) - x) - y]$ into $\circ((x+y)-(x+y))$, then further simplifies it to 0. 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." As we will see in a few paragraphs, in this particular case, 0 is a compliant result, according to the C standard.

If the code was using not only additions but also multiplications, we could experience yet another source of nonreproducibility, due to the potential use of the *fused multiply-add* operator. As an example, let us see how a C compiler may implement the floating-point evaluation of `a * b + c * d`. When an FMA is available, the various compilation schemes may lead to three different results: $\circ(\circ(a \cdot b) + \circ(c \cdot d))$, $\circ(a \cdot b + \circ(c \cdot d))$, and $\circ(c \cdot d + \circ(a \cdot b))$. Many more examples of strange floating-point behaviors can be found [Mon08, MBdD+18].

```
void test(double x, double y) {
  const double y2 = x + 1.0;
  if (y != y2) printf("error\n");
}

int main() {
  const double x = .012;
  const double y = x + 1.0;
  test(x, y);
  return 0;
}
```

Figure 4.8 – Example from GCC bug #323

The C standard

As surprising as it may seem, discrepancies caused by the choice of the target format for each operation or by the use of the FMA operation are allowed by the ISO C standard, which leaves much freedom to the compiler in the way it implements floating-point computations [ISO11]. That way, vastly incompatible architectures could be supported at the time the C programming language was first designed.

> "The values of operations with floating operands [...] are evaluated to a format whose range and precision may be greater than required by the type." (Characteristics of floating types [ISO11, §5.2.4.2.2])

While Annex F of the C standard allows a compiler to advertise compliance with IEEE-754 floating-point arithmetic if it supports a specified set of features, none of these features reduces the leeway compilers have in choosing intermediate formats.

Note that this optimization opportunity also applies to the use of an FMA operator for computing the expression $a \cdot b + c$, as the intermediate product is then performed with a much greater precision. This also explains why the result obtained at level `-Ofast` in Table 4.1 is actually allowed.

Unfortunately, some compilers interpret the standard in an even more relaxed way: values of local variables that are not spilled to memory might preserve their extended range and precision. Consider the code of Figure 4.8 as another example of this miscompilation. It is adapted from GCC's infamous bug 323.[6] For an x86 32-bit target at optimization level `-O1`, all versions of GCC prior to 4.5 miscompile this code as follows: the expression `x + 1.0` in function `test` is computed in extended precision, as allowed by C, but the compiler omits to round it back to double precision when assigning to y_2, as prescribed by the C standard. Consequently, y and y_2 compare different, while they must be equal according to the C standard.

To summarize the situation in a few sentences, the IEEE-754 standard precisely specifies what floating-point operations do (let us ignore NaNs for now) and modern processors tend

[6] "Optimized code gives strange floating-point results", `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=323`

to be compliant with it. But, for historical reasons, the language standards are usually not as strict and allow for multiple interpretations of arithmetic expressions. Moreover, compiler optimizations might lead to even more runtime behaviors, either because of a bug in their implementation or for the sake of runtime performance. As a consequence, it seems like we should either give up on clever uses of floating-point arithmetic, *e.g.*, Fast2Sum operators, or we should spend time verifying that the machine code matches the source code.[7]

CompCert C

A solution to the later issue is the use of a *verified compiler*. Verifying a compiler requires two parts. First, one should give some semantics to the source language and to the target language. Second, one should state and prove that the compiler preserves the semantics of a program when compiling from the source language to the target language. The CompCert C compiler, written in Coq, is such a verified compiler [Ler09]. Its source language is a subset of the C99 language. CompCert's specification of its semantics hopefully complies with the standard. Its target language is Assembly from one of four supported architectures: ARM, PowerPC, RISC-V, x86. CompCert's specification of their semantics is hopefully faithful to the way these processors behave.

CompCert associate *observable behaviors* with every program. These behaviors include normal termination, divergence (the program runs forever), and encountering an undefined behavior (such as an out-of-bounds array access). They also include traces of all input/output operations performed by the program: calls to I/O library functions (such as `printf`) and accesses to `volatile` memory locations. For example, CompCert's C semantics predicts that the Fast2Sum program shown at the start of the chapter has a single behavior, namely, normal termination with exit code 0 and a trace consisting of a single observable event: a call to `printf` with argument `0x1.fffffp-54`. The following theorem has been formally verified [Ler09].

Theorem 4.1 (Semantic preservation) *Let S be a source C program. Assume that S is free of undefined behaviors. Further assume that the CompCert compiler, invoked on S, does not report a compile-time error, but instead produces executable code E. Then any observable behavior of E is one of the possible observable behaviors of S.*

In early versions of CompCert, the floating-point part of the formalization was not constructed, but only axiomatized: the type of floating-point numbers was an abstract type, the arithmetic operations were just declared as functions but not realized, and the algebraic identities exploited during code generation were not verified, 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 dependency on the host platform when performing numerical computations at compile time, *e.g.*, input and output of floating-point literals.

These issues were fixed in CompCert 1.11. To do so, I extended Flocq with a bit-level formalization of IEEE-754 arithmetic (see Section 4.3.2), Sylvie Boldo proved helper lemmas, Jacques-Henri Jourdan plugged Flocq into CompCert, and Xavier Leroy verified some higher-level algorithms, *e.g.*, conversion to and from integer types [BJLM13, BJLM15]. We can now

[7] For example, when developing the Boost.Interval library, the analysis of the Assembly code generated by GCC led me to report various bugs, as its optimization passes were not conservative enough in presence of directed rounding modes.

state with a high level of confidence that the following floating-point properties are satisfied by C programs compiled using CompCert:

- The `float` and `double` types are mapped to the *binary32* and *binary64* formats, respectively. Extended-precision floating-point numbers are not supported: the `long double` type is either unsupported or mapped to *binary64*, depending on a compiler option.
- Conversion to a floating-point type, either explicit (*type casts*) or implicit (at assignment, parameter passing, and function return), always rounds the floating-point value to the given floating-point type, discarding excess precision.
- Reassociation of floating-point operations, or *contraction* of several operations into one (*e.g.*, a multiplication and an addition being contracted into a fused multiply-add) are prohibited. On target platforms that support an FMA instruction, CompCert makes it available as a compiler built-in function. So, the compiled code uses FMAs, only if the source program does so explicitly.
- All intermediate floating-point results in expressions are computed with the precision that corresponds to their static C types, *i.e.*, the largest of the precisions of their arguments.
- All floating-point computations round to nearest, tie breaking to even, except conversions from floating-point numbers to integers, which round toward zero. The CompCert formal semantics make no provisions for programmers to change rounding modes at run-time.
- Floating-point literal constants are also rounded to nearest, tie breaking to even.

4.3.2 Bit-level formalization of IEEE-754 arithmetic

To support floating-point arithmetic in CompCert, part of the IEEE-754 standard had to be formalized. This section focuses on the side of the formalization that pertains to Flocq. For more details on how the CompCert side, see the article and the book [BJLM15, BM17]. Note that we are only considering binary formats, not decimal ones.

Binary formats

In Section 3.2, Flocq formalized floating-point numbers by real numbers both representable as $m \cdot \beta^e$ and part of a format. The formats of Flocq that most closely match IEEE 754 are FLT, but they are far from sufficient. First, they have no upper bound on the exponent range, so they contain larger elements than the *binary32* and *binary64* formats. Second, as subsets of $\mathbb{R}$, they provide none of the exceptional values: signed zeroes, infinities, NaNs.

Therefore, Flocq introduces an inductive type `binary_float` that represents all theses cases and properly restricts the possible values. It is parametrized by the precision (including the implicit bit, *e.g.*, $\varrho = 53$ for *binary64*) and by the exponent of the first non-representable power of 2 (*e.g.*, $e_{\max} = 1024$ for *binary64*). The constructors of the inductive type represent zeroes, infinities, NaNs, and finite numbers. For each of them, the first argument is the sign, which is *true* when negative.

```
Inductive binary_float (prec emax : Z) : Set :=
  | B754_zero     (s : bool) : binary_float prec emax
  | B754_infinity (s : bool) : binary_float prec emax
  | B754_nan      (s : bool) (pl : positive) :
    nan_pl prec pl = true -> binary_float prec emax
  | B754_finite   (s : bool) (m : positive) (e : Z) :
    bounded prec emax m e = true -> binary_float prec emax.
```

For NaNs, the other arguments of the constructor are the payload and a proof that the payload fits in the format. This property is expressed using the following function, with `(Zpos (digits2_pos pl))` denoting the number of bits of the payload `pl`, which must be smaller than ϱ.

```
Definition nan_pl (prec : Z) (pl : positive) :=
  Zlt_bool (Zpos (digits2_pos pl)) prec.
```

For finite numbers, the other arguments of the constructor are the integer significand and the exponent, as well as a proof that they are canonical with respect to the FLT format of precision ϱ and of minimal exponent $e_{\min} = 3 - e_{\max} - \varrho$. The following function expresses (in a computational way) that a pair (m, e) is in canonical form, since the term `(Zpos (digits2_pos m) + e)` computes the magnitude of the number $m \cdot 2^e$.

```
Definition canonical_mantissa (prec emax : Z) (m : positive) (e : Z) : bool :=
  let emin := (3 - emax - prec)%Z in
  Zeq_bool (FLT_exp emin prec (Zpos (digits2_pos m) + e)) e.

Definition bounded (prec emax : Z) (m : positive) (e : Z) : bool :=
  andb (canonical_mantissa prec emax m e) (Zle_bool e (emax - prec)).
```

An important function is `B2R`, which turns a binary floating-point number into the real value it represents (or zero in case of exceptional value). The `Bsign` function returns whether the sign of a floating-point number is negative. A few Boolean functions make it possible to get the category of a floating-point number: `is_finite` (either zero or finite), `is_finite_strict` (only finite, no zero), `is_nan`. Note that, except for $+0$ and -0, no two finite binary floating-point numbers can represent the same real number. As a consequence, the `B2R` function is injective for nonzero finite numbers.[8] In other words, to prove that two such numbers are equal, we just have to prove that the real values they represent are equal and then apply the following theorem.

```
Theorem B2R_inj :
  forall (prec emax : Z) (x y : binary_float prec emax),
  is_finite_strict prec emax x = true ->
  is_finite_strict prec emax y = true ->
  B2R prec emax x = B2R prec emax y ->
  x = y.
```

[8] We do not assume any kind of proof irrelevance here. Not only are the significands and exponents equal, but even the proofs that they are canonical are provably equal.

Bit-level representation

C programs might fiddle with the memory representation of floating-point numbers. So, to capture the semantics of such programs, we also need some conversion functions from floating-point numbers of type `binary_float` to integers, and *vice versa*. Even if we were to ignore such programs, the compiler still needs a way to output floating-point literals as integers in the Assembly code.

These two conversion functions have the following types. They are parametrized by the bit-width of the mantissa (not counting the implicit bit, *e.g.*, $mw = 52$ for *binary64*), and the bit-width of the exponent (*e.g.*, $ew = 11$ for *binary64*). The corresponding format parameters are $\varrho = mw + 1$ and $e_{\max} = 2^{ew-1}$.

```
Definition bits_of_binary_float :
  forall mw ew : Z,
  let prec := (mw + 1)%Z in
  let emax := (2 ^ (ew - 1))%Z in
  binary_float prec emax -> Z.

Definition binary_float_of_bits :
  forall mw ew : Z,
  let prec := (mw + 1)%Z in
  let emax := (2 ^ (ew - 1))%Z in
  (0 < mw)%Z -> (0 < ew)%Z -> (prec < emax)%Z ->
  Z -> binary_float prec emax.
```

Note that `binary_float_of_bits` is parametrized, not only by mw and ew, but also by proofs that they are positive, as well as a proof that $\varrho < e_{\max}$. That is mostly an artifact of the way the verification was performed and could presumably be worked around. Yet, it means that the formalism might not be suited for very small formats, *e.g.*, a *binary8* format with 1 bit of sign, 3 bits of biased exponent, and 4 bits of trailing significand.

It is difficult, *a priori*, to know whether these two functions comply with the IEEE-754 standard. Fortunately, since we need them to implement a compiler, they are effective. Thus, we could test them on various bit patterns to convince ourselves that they are correct. Another way to increase the confidence in their implementation is to prove that they at least have consistent definitions. (In the following statements, `mw`, `ew`, and their requirements, have been removed for the sake of readability.)

```
Theorem bits_of_binary_float_range:
  forall x : binary_float,
  (0 <= bits_of_binary_float x < 2 ^ (mw + ew + 1))%Z.

Theorem binary_float_of_bits_of_binary_float :
  forall x : binary_float,
  binary_float_of_bits (bits_of_binary_float x) = x.

Theorem bits_of_binary_float_of_bits :
  forall x : Z,
  (0 <= x < 2 ^ (mw + ew + 1))%Z ->
  bits_of_binary_float (binary_float_of_bits x) = x.
```

Arithmetic operators

Finally, let us talk about the arithmetic operators. In Section 3.2.3, we have already explained how we could implement division for an arbitrary format as a function that takes two positive numbers and returns a number as well as a position of the infinitely precise result with respect to this number. Thus, it is now a matter of turning such a function into one that supports all the exceptional inputs and outputs. First of all, we need an inductive type `mode` that enumerates the five rounding directions mandated by the IEEE-754 standard [IEE08, §4.3].

```
Inductive mode : Set :=
  mode_NE | mode_ZR | mode_DN | mode_UP | mode_NA.
```

All the arithmetic operators take an argument of this type, in addition to the input numbers. They also take a function that characterizes how an operator behaves with respect to NaNs. This function is called whenever the operator needs to output a NaN. It takes the two original inputs as arguments and returns a NaN. The reason for such a function is that the IEEE-754 standard is underspecified when it comes to NaNs. Indeed, it tells when one should be produced, but it does not tell what its sign and payload should be, except that the payload should be propagated from one of the input NaNs, if any.

In the first version of this work, there was only one NaN per floating-point type [BJLM13]. Thus, the formalism was much simpler, as there was no need to ever decide which NaN an operator returns. But this also meant that the function `bits_of_binary_float` was partially specified. Indeed, in case of a NaN, only the exponent bits are perfectly known. Having undefined bits was not in the spirit of CompCert, though, so the second version of the work made it possible to define architecture-dependent arithmetic operators [BJLM15]. This led to the following signature for the division operator.

```
Definition Bdiv :
  forall prec emax : Z,
  (0 < prec)%Z -> (prec < emax)%Z ->
  (binary_float prec emax -> binary_float prec emax ->
     {n : binary_float prec emax | is_nan prec emax n = true}) ->
  mode -> binary_float prec emax -> binary_float prec emax ->
  binary_float prec emax.
```

At first glance, its implementation shown on Figure 4.9 is rather straightforward. One exhaustively handles all the combination of input categories and acts accordingly. Note that `build_nan` is a contrived function, but it can be understood as being just a synonymous for the `B754_nan` constructor. When both arguments are finite numbers, we call the division operator from Section 3.2.3. Then, we round its result according to the mode. Finally, we check whether the exponent exceeds $e_{\max} - \varrho$. If so, we return an infinity or one of the largest floating-point numbers, depending on the sign and the rounding mode.

The specification does not tell much about the behavior in case of exceptional inputs. But, since the implementation is effective, one can just execute it to check what happens. The specification is much more interesting for the finite case (characterized by the fact that the real value of y is nonzero). Indeed, it is based on the IEEE-754 motto [IEE08, §4.3]: "Each operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result [...]."

First, the specification introduces a few abbreviations. It denotes z the result of `Bdiv` applied to the inputs x and y. It denotes r the axiomatic rounding (see Section 3.2.2) of the

```
Definition Bdiv prec emax ... m div_nan x y :=
  match x, y with
  | B754_nan _ _ _, _ | _, B754_nan _ _ _  => build_nan (div_nan x y)
  | B754_infinity sx, B754_infinity sy     => build_nan (div_nan x y)
  | B754_infinity sx, B754_finite sy _ _ _ => B754_infinity (xorb sx sy)
  | B754_finite sx _ _ _, B754_infinity sy => B754_zero (xorb sx sy)
  | B754_infinity sx, B754_zero sy         => B754_infinity (xorb sx sy)
  | B754_zero sx, B754_infinity sy         => B754_zero (xorb sx sy)
  | B754_finite sx _ _ _, B754_zero sy     => B754_infinity (xorb sx sy)
  | B754_zero sx, B754_finite sy _ _ _     => B754_zero (xorb sx sy)
  | B754_zero sx, B754_zero sy             => build_nan (div_nan x y)
  | B754_finite sx mx ex _, B754_finite sy my ey _ => ...
  end.

Theorem Bdiv_correct :
  forall (prec emax : Z) (Hp : (0 < prec)%Z) (Hm : (prec < emax)%Z)
    (div_nan : ...) (m : mode) (x y : binary_float prec emax),
  let emin := (3 - emax - prec)%Z in
  B2R prec emax y <> 0%R ->
  let r := round radix2 (FLT_exp emin prec) (round_mode m)
    (B2R prec emax x / B2R prec emax y) in
  let z := Bdiv prec emax Hp Hm div_nan m x y in
  let sign := xorb (Bsign prec emax x) (Bsign prec emax y) in
  if Rlt_bool (Rabs r) (bpow radix2 emax)
  then  (* finite case *)
    B2R prec emax z = r /\
    is_finite prec emax z = is_finite prec emax x /\
    (is_nan prec emax z = false -> Bsign prec emax z = sign)
  else  (* overflow *)
    B2FF prec emax z = binary_overflow prec emax m sign.
```

Figure 4.9 – Implementation and specification of division.

infinitely precise quotient of the real values of x and y. Second, it compares $|r|$ to $2^{e_{\max}}$. If it is smaller, then the real value of z is r. Moreover, its sign is the exclusive OR of the signs of x and y.[9] Otherwise, if r is too large to be represented as a finite number, the behavior is described by the `binary_overflow` function.

All the arithmetic operators follow the same scheme of implementation and specification. The latter is quite verbose, but it is so close to the IEEE-754 standard that it gives a very high level of confidence in the correctness of the implementation. Finally, since all the functions have an effective implementation, they can be used in CompCert, not only to specify the semantics of the source and target languages, but also to actually implement some optimizations such as *constant propagation*. For example, by using constant propagation and function inlining, CompCert is able to turn the body of function `g` in the following C code into just "`return 0x1.5555555555555p-2;`".

```
inline double f(double x) {
  if (x < 1.0) return 1.0;
  else return 1.0 / x;
}
double g(void) {
  return f(3.0);
}
```

As an aside, note that the division operator described here is not the only one provided by Flocq. Indeed, when parsing a decimal floating-point literal, the compiler needs to turn it into a *binary32* or *binary64* number. Fortunately, at no point does Flocq's implementation depend on the fact that the inputs have some limited precision or exponent. Thus, when converting $m \cdot 10^e$ with e negative, the compiler first computes the integer 10^{-e} exactly. It then asks Flocq what the floating-point quotient of $m \cdot 2^0$ by $10^{-e} \cdot 2^0$ is (with rounding to nearest, ties breaking to even).

4.3.3 Assessment

Because we chose some strict semantics of floating-point operators (bit-precise results, even for NaNs) and the compiler has to preserve them, many optimization opportunities are lost by CompCert. For example, optimizing $x \oplus (-y)$ to $x \ominus y$ might be incorrect when y is a NaN. If we wish to keep the strict semantics, there are two main ways in which to improve the compiler. First, the compiler could perform some naive *value range propagation* on floating-point values, so as to know when some exceptional values are impossible. For example, when encountering a conditional "`if (0. < x + y) {...}`", the compiler would know that, inside the block, neither x nor y are NaN or negative infinity. Second, the compiler could analyze the code to see whether the sign and payload of exceptional values can ever be observed. For example, if the final result of a floating-point computation is only used in a comparison, then neither the sign of zero nor the payload of NaN matter. In both cases, this opens the way to applying many transformations on floating-point expressions.

Regarding the effectiveness of Flocq's floating-point operators, there is an interesting issue. For finite numbers and NaNs, the `binary_float` type embeds proofs that both significand and exponent are properly sized. Due to Coq's formal system, it means that the datatype

[9] This only matters when the infinitely precise quotient is so small that it got rounded to zero. In all the other cases, the sign of z can be trivially inferred from the implementation.

physically contains objects representing these proofs. In particular, when performing arithmetic computations, these proof terms might grow large. For the compiler, this is not an issue, since Coq's *extraction* mechanism assumes that the proof terms are non-informative and erases them from the extracted OCaml code as if they were singleton objects [Let02]. But, when performing computations using Coq's VM engine, the proof terms will carry the whole history of computations.[10] Since these terms are basically proofs of Boolean equality, we could insert identity functions that erase these proof terms by recomputing the Boolean values from scratch. Unfortunately, this overhead would also be paid in an extracted program.

Another solution would be to have some kind of degraded mode for the VM. It would consider proof terms to be non-informative and thus would not store them, but it would not consider them to be singleton either and would error out if one of them is still apparent at the end. Thus, such a degraded mode would make it impossible to obtain a floating-point value at the end of a computation, but it would be possible to obtain its integer representation. It would also be possible to get the result of comparing two floating-point values.

4.4 Homogeneous geometric predicates

Let us consider a very different kind of mathematical library: the one we can find in computational geometry. Such libraries provide geometric predicates that qualify the relative positioning of points. For instance, some predicate might characterize whether a point is inside the sphere that goes through four other points. While such a predicate might be implemented using floating-point arithmetic for efficiency reasons, there is a major difference with most other floating-point algorithms, as the result of the predicate does not approximate a real number. Indeed, it approximates a discrete value: either the point is inside the sphere, or it is on the surface, or it is outside. As such, there is no straightforward notion of closeness for the result of such a predicate. If the predicate gets it wrong, this might invalidate basic geometric theorems on which algorithms from computational geometry rely [KMP+04]. So, the goal is to devise implementations that always return the correct result [YD95]. More precisely, we will focus on devising floating-point implementations that either compute the exact result or answer that they do not know what the exact result is. Such an implementation can then act as the first stage of an exact predicate as it is done in the CGAL library [FGK+00].

When Sylvain Pion and I originally devised such floating-point predicates, we made an informal proof of their correctness using Gappa [MP05, MP07]. While Gappa generated a formal proof, this proof corresponded to a modified algorithm and it would have been incorrect when applied to the original algorithm. The results of the Gappa's analysis, however, were correct, thanks to some meta-arguments on the way Gappa found them. I later revisited this work and turned the informal proofs into fully verified Coq proofs [BM17, §6.6].

We will illustrate the approach using the simplest of the geometric predicates, `orient2d`, which characterizes on which side of a line formed by two points a third point of the 2D plane is located, as shown in Figure 4.10. Section 4.4.1 describes how a naive floating-point implementation of `orient2d` behaves. Section 4.4.2 then shows how to devise a reliable floating-point algorithm and how to formally verify it.

[10] This is not entirely accurate. For example, since division does not care about the proofs that its inputs have the expected size, the history gets truncated each time a division is performed.

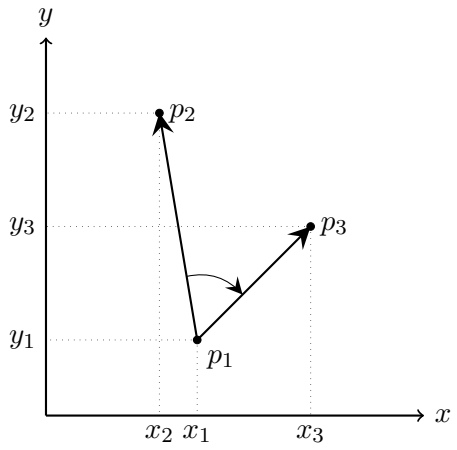


Figure 4.10 – Clockwise orientation of three points p_1, p_2, and p_3.

4.4.1 Naive floating-point implementation

Using the coordinates of these points, the predicate can be expressed as the sign of a determinant:

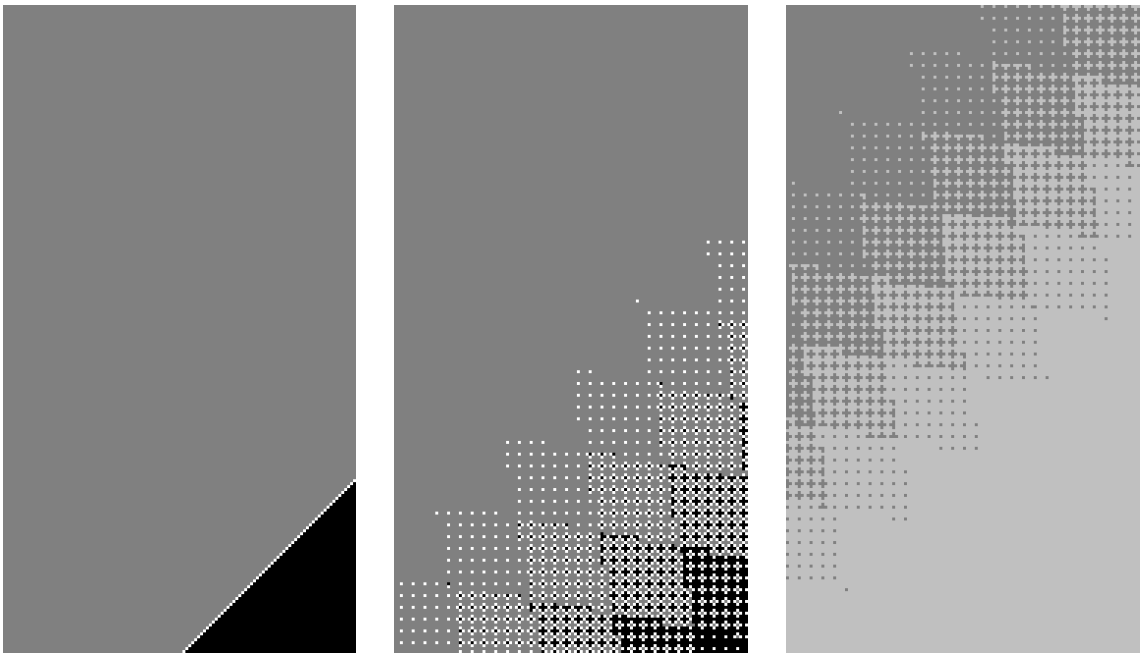
$$\texttt{orient2d}(p_1, p_2, p_3) = \operatorname{sign} \begin{vmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{vmatrix}.$$

When the three points, p_1, p_2, and p_3, are aligned, the determinant is zero. When they are oriented clockwise, it is negative. When they are counterclockwise, it is positive. Its sign can be computed using four exact subtractions to get the two vectors, then two exact multiplications and one exact subtraction to compute the 2×2 determinant, and finally a comparison to zero. While an implementation using an exact arithmetic is possible, it is slow and should only be used as a last resort. So, we want to use floating-point arithmetic to perform a fast computation of the sign and to properly detect when the computed sign is possibly incorrect.

Let us assume that the input coordinates are representable as *binary64* floating-point numbers. If floating-point operations were infinitely precise, one would get a fast and reliable implementation by using a straightforward translation to C code:

```
int orient2d(double x1, double y1, double x2,
             double y2, double x3, double y3)
{
  double det = (x2 - x1) * (y3 - y1)
             - (x3 - x1) * (y2 - y1);
  if (det < 0) return NEGATIVE;
  if (det > 0) return POSITIVE;
  return ZERO;
}
```

Since floating-point operations are usually not exact, round-off errors can cause *det* to have an unexpected sign. Note that the round-off error of the last floating-point subtraction does not matter, since we are only interested in the sign of the result and floating-point subtraction preserves the sign of the exact computation, as both 0 is in the format and the rounding is



(a) exact arithmetic (b) floating-point arithmetic (c) floating-point arithmetic with filtered results

Figure 4.11 – Results of the `orient2d` algorithm depending on the arithmetic. White points are computed as aligned, black points as oriented clockwise, and dark gray points as counterclockwise. Light gray points in Figure 4.11c mark possibly incorrect results.

monotone. The round-off errors of the six first operations, however, are propagated and can cause the final sign to be incorrectly computed. Not only might *det* be computed as zero even if the points are not aligned, but its sign might even be the opposite of the exact one.

Figure 4.11 illustrates the situation. Point p_1 visits all the points with *binary64* coordinates around $(1.1, 1.1)$, which is located in the lower-right part of every picture. Points p_2 and p_3 are fixed at positions $p_2 = (3.125, 3.125 - 2^{-51})$ and $p_3 = (3.25 + 2^{-51}, 3.25)$. The distance between two neighboring points is 2^{-52}. Each of the three pictures shows the sign of the determinant depending on the position of p_1. When p_1 is on some diagonal line (visible on Figure 4.11a), the three points are aligned, so the sign should be zero. In the upper-left corner of every picture, the three points are oriented counterclockwise, while in the lower-right corner, they are oriented clockwise.

Figure 4.11a shows the ideal situation: the arithmetic operations are performed exactly and the computed sign matches the orientation of the three points. Figure 4.11b shows the situation with *binary64* arithmetic: some orientations outside the diagonal line are incorrectly computed as aligned (white points); there are even some cases where the sign is computed as positive while it should be negative, and conversely.

Our goal is to devise a floating-point algorithm that detects when the previous result might be incorrect. This algorithm is presented in Section 4.4.2 and shown in Figure 4.12. The light gray area in Figure 4.11c shows the points for which the algorithm estimates that

the result is possibly incorrect. In that case, a correct but slower code has to be run. Notice that all the incorrectly computed orientations in Figure 4.11b are properly detected. Notice also that some orientations, while correctly computed, are detected as unreliable, which will cause the slower code to be called.

Given a static bound Δ on the absolute error between the computed floating-point value *det* and the ideal value *Det*, we could easily turn the naive algorithm into a reliable one. Indeed, if the absolute value of the computed result *det* is larger than Δ, then *det* has the same sign as *Det*. We would thus obtain an algorithm with the following properties: sound (if the bound is correct), useful (if the bound is tight), and fast (hardly more floating-point operations). Its C code would look as follows.

```
int orient2d(double x1, double y1, double x2,
             double y2, double x3, double y3)
{
  double det = (x2 - x1) * (y3 - y1)
             - (x3 - x1) * (y2 - y1);
  double delta = ...;
  if (det < -delta) return NEGATIVE;
  if (det > +delta) return POSITIVE;
  return UNKNOWN;
}
```

Unfortunately, we cannot compute a tight bound *a priori*. Indeed, a bound on the absolute error would not be tight since it would have to work for large inputs and thus would be useless for small inputs. A bound on the relative error would not make any sense either, since we are interested in the sign of *det*. Indeed, as its sign can be wrong, it means that the relative error of the algorithm is unbounded.

Instead of a static bound, let us compute a bound on the absolute error between *det* and *Det* at execution time. Evaluating *det* using interval arithmetic would provide us with such a bound, but we want some faster mechanism that does not involve directed rounding modes. First, let us consider a more specific problem. We ask Gappa (see Section 4.2.1) what the bound Δ_1 on the absolute error is, when all the differences $|x_2 - x_1|$, $|y_2 - y_1|$, $|x_3 - x_1|$, and $|y_3 - y_1|$ are bounded by 1. The coordinates do not have to be bounded. For the *binary64* format, the tool answers $\Delta_1 = 2^{-51}$.

Now, let us suppose that the differences $|x_2 - x_1|$ and $|x_3 - x_1|$ are no longer bounded by 1, but by 2. Gappa then computes an error bound twice as large. Similarly, if we were to double the range of the differences on the y inputs, we would get a doubled error bound. This behavior can be explained from the shape of the formulas used for forward error analysis (see Section 2.1.4). As a consequence, we could expect the following value Δ to be a general bound on the absolute rounding error:

$$\Delta = \Delta_1 \cdot M \quad \text{with } M = \left\| \begin{matrix} x_2 - x_1 \\ x_3 - x_1 \end{matrix} \right\|_\infty \cdot \left\| \begin{matrix} y_2 - y_1 \\ y_3 - y_1 \end{matrix} \right\|_\infty . \tag{4.2}$$

The vector norm used in this formula is just the maximum of the absolute values, so the computation of Δ would not be too expensive. Indeed, the absolute value is a cheap operation, so the overhead with respect to the straightforward implementation of the predicate would be two maxima and two multiplications. While Δ is defined using exact operations on real

numbers, we will later see how to replace them with floating-point operations. So, with respect to speed, the above definition of Δ is suitable.

Unfortunately, there are several issues on the correctness side. First of all, when the floating-point products of *det* underflow, the value Δ is no longer an upper bound on the absolute round-off error. Indeed, Δ_1 is a value obtained assuming that no underflow occurs. But even for the normal case, one has to be careful when reusing the value Δ_1 computed by Gappa. While this value is correct, it is only meaningful when the input differences are bounded by a power of 2. This is easily seen by asking Gappa the error bound for some other ranges. For a *binary64* format rounded to nearest, Gappa answers $\Delta_1 = 2^{-51}$ when the input difference ranges are $[-1; 1]$. If we extend the ranges to $[-1.01; 1.01]$, Gappa answers $\Delta_{1.01} = 2^{-49.98}$. So a slight modification of the inputs causes the error bound computed by Gappa to be twice as large. In a way, the bound Δ_1 computed by Gappa is too tight for our purpose.

Choosing a suitable value of Δ_1 is not sufficient; we also have to find under which condition Equation (4.2) gives a proper bound on the absolute round-off error. We would like to obtain the following property with M as in Equation (4.2):

$$2^{-1022} \leq M \Rightarrow |det - Det| \leq 2^{-49.99\ldots} \cdot M. \tag{4.3}$$

Our approach should also be generic enough so that it can be applied to more complicated geometric predicates. We will follow the structure of the expressions, inductively computing for every subexpression a bound on its round-off error. This bound will be computed as the product between some expression (ultimately M) and a constant factor (ultimately Δ_1). The property satisfied by this bound will be expressed as a relation $\mathcal{R}$, from which we will deduce Equation (4.3).

4.4.2 Reliable floating-point implementation

The way $\mathcal{R}$ is built is skipped for conciseness; see the book for more details [BM17, §6.6.2]. The basic idea is that we want an enclosure that looks like $|\tilde{x} - x| \leq \delta \cdot m$, so we design a relation that embeds it yet can be propagated along all the arithmetic operations needed for `orient2d`. This gives the following relation:

$$\mathcal{R}(c, \tilde{x}, x, m, b, \delta) \quad \stackrel{\text{def}}{=} \quad \mu \cdot \varepsilon^{-1} \leq c \Rightarrow \begin{cases} b \geq 1, \\ \delta \geq 0, \\ |x| \leq b \cdot m, \\ |\tilde{x} - x| \leq \delta \cdot m. \end{cases}$$

Note that the values b and δ are numerical values, while c and m are symbolic expressions. The constants ε and μ depend on the floating-point format $\mathcal{F}$ and on the rounding mode. For *binary64* and rounding to nearest, we have $\varepsilon = 2^{-53}$ and $\mu = 2^{-1075}$.

We start the forward error analysis with the four subtractions of coordinates. To express the relation between the rounded differences and their exact values, we take advantage of the fact that the relative error of the floating-point sum is always bounded, even if the result is in the subnormal range: $|\circ(u - v) - (u - v)| \leq \varepsilon \cdot |u - v|$. This can be turned into the following instance of the relation:

$$\forall u, v \in \mathcal{F}, \quad \mathcal{R}(\mu \cdot \varepsilon^{-1}, \circ(u - v), u - v, |u - v|, 1, \varepsilon).$$

As always, we split floating-point operators into an infinitely precise operator and a rounding operator. Let us start with the rounding operator. Most of the complexity of $\mathcal{R}$ comes from it. The relation is propagated using the following property:

$$\mathcal{R}(c, \tilde{x}, x, m, b, \delta) \quad \Rightarrow \quad \mathcal{R}(\min(c, m), \circ(\tilde{x}), x, m, b, \delta + (b + \delta) \cdot \varepsilon).$$

The propagation rules for addition and multiplication are straightforward. Their consequent are verified using forward error analysis. Given

$$\mathcal{R}(c_u, \tilde{u}, u, m_u, b_u, \delta_u) \wedge \mathcal{R}(c_v, \tilde{v}, v, m_v, b_v, \delta_v),$$

we can deduce

$$\begin{array}{llllll} \mathcal{R}(\min(c_u, c_v), & \tilde{u} + \tilde{v}, & u + v, & \max(m_u, m_v), & b_u + b_v, & \delta_u + \delta_v), \\ \mathcal{R}(\min(c_u, c_v), & \tilde{u} \cdot \tilde{v}, & u \cdot v, & m_u \cdot m_v, & b_u \cdot b_v, & \delta_u \cdot b_v + b_u \cdot \delta_v + \delta_u \cdot \delta_v). \end{array}$$

All the formulas above use real numbers to represent the bounds b and δ. To simplify the proof process in Coq, it is better to formalize and verify all these formulas using unbounded floating-point numbers (or rational numbers), so that the proof assistant will automatically compute the values of b and δ of the conclusion of a formula from its hypotheses. For instance, the statement of the formula for the sum is expressed as follows, with the real addition to sum expressions and an exact addition `Fplus` of floating-point numbers to sum the bounds.

```
Rel c u1 v1 m b1 d1 -> Rel c u2 v2 m b2 d2 ->
Rel c (u1+u2) (v1+v2) m (Fplus b1 b2) (Fplus d1 d2).
```

When applied to the `orient2d` predicate using *binary64* arithmetic with rounding to nearest, we get the following relation (with an approximated δ for the sake of readability) from which we will be able to deduce Equation (4.3):

$$\mathcal{R}(M, det, Det, M, 2, 2^{-49.99\cdots}) \quad \text{with } M = \left\| \begin{array}{c} x_2 - x_1 \\ x_3 - x_1 \end{array} \right\|_\infty \cdot \left\| \begin{array}{c} y_2 - y_1 \\ y_3 - y_1 \end{array} \right\|_\infty. \tag{4.4}$$

We need to tackle one last issue before obtaining the reliable implementation shown in Figure 4.12. Indeed, in Equation (4.3), the values M and $2^{-49.99\cdots} \cdot M$ are expressed using exact arithmetic, but in the implementation, floating-point computations are used. Thus, the constant 2^{-1022} has to be increased to take into account any underestimation while approximating M. Similarly, the constant $\delta = 2^{-49.99\cdots}$ has to be increased to take into account any underestimation while approximating $\delta \cdot M$. The relative error between the floating-point value m and the infinitely precise value M is bounded by $\theta_3 = (1 + 2^{-53})^3 - 1$ (two subtractions and one multiplication), assuming the last multiplication does not underflow. So, we have $M \cdot (1 - \theta_3) \leq m \leq M \cdot (1 + \theta_3)$. By choosing $C_1 = \triangle(2^{-1022} \cdot (1 + \theta_3))$ and $C_2 = \triangle(\delta \cdot (1 - \theta_3)^{-1})$, the following implications hold:

$$\begin{array}{rcl} m \geq C_1 & \Rightarrow & M \geq 2^{-1022}, \\ |det| > \circ(C_2 \cdot m) & \Rightarrow & |det| > \delta \cdot M. \end{array}$$

Note that the inequality $m \geq C_1$ also implies that the last multiplication performed when computing m did not underflow, so we do not need any special check to exclude that case.

There could still be an issue with *det* ending up either infinite or NaN. The NaN case is simple: both final comparisons will fail, so the end result will be *unknown*. The infinite case is

```
int orient2d(double x1, double y1, double x2,
             double y2, double x3, double y3)
{
  double dx1 = x2 - x1;
  double dx2 = x3 - x1;
  double dy1 = y2 - y1;
  double dy2 = y3 - y1;
  double nx = fmax(fabs(dx1), fabs(dx2));
  double ny = fmax(fabs(dy1), fabs(dy2));
  double m = nx * ny;
  if (m < 0x1.0000000000002p-1022) return UNKNOWN;
  double det = dx1 * dy2 - dx2 * dy1;
  double delta_m = 0x1.0000000000003p-50 * m;
  if (det < -delta_m) return NEGATIVE;
  if (det > +delta_m) return POSITIVE;
  return UNKNOWN;
}
```

Figure 4.12 – Reliable implementation of `orient2d`.

a bit more contrived. Let us see where the infinity comes from. First, if neither floating-point products $dx_1 \cdot dy_2$ nor $dx_2 \cdot dy_1$ are infinite yet their difference is infinite, they have to be of opposite signs. So, the sign of the computed determinant is necessarily correct in that case. Second, if either floating-point product is infinite, then m is also infinite since it is larger than both products in absolute value. Thus, `delta_m` is $+\infty$, which means that the last two comparisons will fail, leading to an *unknown* result. Therefore, the algorithm is robust not only to round-off errors but also to overflow issues.

In Equation (4.4), the value of c ends up being equal to M after simplification. So, a simpler definition of $\mathcal{R}$ could have been used to verify `orient2d`. More complicated geometric predicates, however, need the complete definition. Let us illustrate it with the predicate that characterizes the location of a point p_1 of the 2D plane with respect to the circumscribed circle of three other points p_2, p_3, and p_4:

$$\texttt{incircle2d}(p_1, p_2, p_3, p_4) = \operatorname{sign} \begin{vmatrix} x_2 - x_1 & y_2 - y_1 & (x_2 - x_1)^2 + (y_2 - y_1)^2 \\ x_3 - x_1 & y_3 - y_1 & (x_3 - x_1)^2 + (y_3 - y_1)^2 \\ x_4 - x_1 & y_4 - y_1 & (x_4 - x_1)^2 + (y_4 - y_1)^2 \end{vmatrix}.$$

The above approach produces the following property for *binary64* computations:

$$2^{-1022} \leq \min(X^2, Y^2, M) \Rightarrow |det' - Det'| \leq 2^{-45.99\ldots} \cdot M$$
$$\text{with } X = \left\| \begin{matrix} x_2 - x_1 \\ x_3 - x_1 \\ x_4 - x_1 \end{matrix} \right\|_\infty,\ Y = \left\| \begin{matrix} y_2 - y_1 \\ y_3 - y_1 \\ y_4 - y_1 \end{matrix} \right\|_\infty,\ M = X \cdot Y \cdot \max(X^2, Y^2).$$

4.4.3 Assessment

There are a few parts of the analysis that are a bit naive. For example, bounds are scaled using infinitely precise values, yet it would not take much effort to scale them using actually computed values. That way, we would not have to degrade the bounds at the end to account for additional rounding errors. This is especially apparent when initializing the relation, since both $|\circ(u-v)-(u-v)| \leq \varepsilon \cdot |u-v|$ and $|\circ(u-v)-(u-v)| \leq \varepsilon \cdot |\circ(u-v)|$ hold.

There is a much more limiting issue with the algorithm we presented, as discovered by Ozaki *et al* [OBO+16]. Indeed, the orientation of many triples of points is computed as unreliable, due to overflow in the detection code. Yet these triples are nowhere near the dangerous region, so the naive floating-point algorithm does return a correct sign. The algorithm they propose uses a different factor to compute the dynamic error bound. While our factor was $\circ[\max(|x_2-x_1|,|x_3-x_1|)\cdot\max(|y_2-y_1|,|y_3-y_1|)]$, theirs is $|\circ[(x_2-x_1)\cdot(y_3-y_1)+(x_3-x_1)\cdot(y_2-y_1)]|$ plus some correcting term. This has two consequences. First, on instruction sets that lack the `maxNumMag` instruction mandated by the IEEE-754 standard, their factor can be much faster to compute than ours. Second, their factor can be much smaller (hence better) when the initial subtractions give results with widely unbalanced magnitudes, which makes it possible for their algorithm to filter a lot more triples. That being said, they have only studied the `orient2d` predicate. It is unclear whether their approach can be generalized to other geometric predicates, *e.g.*, `incircle2d`, contrarily to ours.

4.5 SMT solvers

Gappa is a solver dedicated to verifying floating-point properties of real-valued expressions. It does so by replicating the kind of reasoning used to prove the correctness of state-of-the-art floating-point hardware and software. It automates the proofs in a highly efficient way, as long as the verification conditions (*e.g.*, generated by Why3) only deal with arithmetic constructs. Unfortunately, some program constructs, *e.g.*, arrays, tend to leak into the verification conditions and obfuscate the arithmetic constructs Gappa relies on. Moreover, as a side-effect of their generation process, verification conditions might be littered with useless lemmas and equalities. All of these issues cause extra work to the user and thus partly defeat the point of automating the verification process.

Contrarily to Gappa, SMT solvers, which are ubiquitous in deductive program verification, are especially designed to handle these issues. They have built-in theories of arrays and congruences, they have specialized algorithms for instantiating lemmas, they depend on SAT solvers for propositional logic, and so on. (See also Section 2.3.1). As for floating-point arithmetic, the SMT-LIB standard includes a formal semantics named FPA [BTRW15].

There are three kinds of techniques for integrating floating-point reasoning in SMT solvers. The first one makes use of *bit blasting*, that is, it eagerly interprets floating-point numbers as bit-vectors and floating-point operations as Boolean circuits. They are then handled by the SAT engine (or the bit-vector engine) of the SMT solver. The second technique consists in lifting the CDCL procedure at the heart of SMT to an abstract algorithm manipulating floating-point numbers as abstract values. For example, real intervals might be used to overapproximate floating-point values. To do so, the Boolean constraint propagation of CDCL is extended with interval constraint propagation and decision steps make a case analysis by interval splitting [BDG+14]. The third technique reduces floating-point numbers to real numbers by means of a rounding operator. Floating-point terms are replaced by equal-valued

exact-arithmetic terms in a fragment of real/integer arithmetic extended by ceiling and floor functions. Rounding is performed using a case analysis to determine the binades of real values [LMRW14].

The first technique is complete, as long as no real arithmetic is involved, which excludes properties on rounding errors. The third technique is complete regarding rounding errors, but only if the only arithmetic operations are addition and multiplication by constants. While we do not aim at completeness, the above limitations are a bit too harsh. So, from what had been learned from the development and usage of Gappa, two other approaches were tried. The objective was to add floating-point support to an SMT solver, namely Alt-Ergo. As in Gappa, floating-point operations are represented by real-valued expressions extended with rounding operators.

4.5.1 Embedding Gappa into Alt-Ergo

SMT solvers that support quantifiers usually work by iterating two phases. First, they check the (un)satisfiability of the current set of quantifier-free literals. Then, if they were not able to conclude, they instantiate the quantified formulas using the current ground terms to produce new literals, and they try again. The difficulty lies in choosing which quantified formulas to instantiate and how. If the correct instances are missing, the SAT and theory solvers will work for naught. If too many instances are added, the SAT and theory solvers might take ages to conclude.

The way Gappa works (see Section 4.2.1) does not fit this framework. Indeed, during its top-down phase, Gappa does not fully instantiate the statements from its theorem database. It is only during the bottom-up phase that the remaining quantifiers will be instantiated with the result of numerical computations.

The first attempt at floating-point support, implemented by Cody Roux with the collaboration of Sylvain Conchon, Mohamed Iguernelala, and me, was to embed a similar mechanism into Alt-Ergo [CMRI12]. This floating-point procedure receives from the main loop of Alt-Ergo the set of terms that have an arithmetic operator or a rounding operator as head symbol. It also receives the set of literals that involve arithmetic, *e.g.*, $u \leq v$. Like Gappa, the procedure tries to instantiate lemmas from the floating-point theory that are deemed relevant for finding interesting bounds on the considered terms. The set of lemmas are therefore internal to the procedure, and computational instances are created when needed, by matching modulo equality on the considered terms. The procedure then checks which lemmas may actually be applied given the bound information on the terms, and applies these lemmas until no new consequences can be created. A simplex-based algorithm then finds the optimal bounds by linear combination of these. Improved bounds may then lead to another round of saturation by application of the instantiated lemmas. If no contradictions (empty bounds) are found, then equalities that were deduced by the simplex algorithm are sent back to the main loop of Alt-Ergo.

Unfortunately, we were not able to solve some performance issues. In particular, Alt-Ergo was slower even in the absence of floating-point operators, since the procedure would be triggered as soon as real operators are present. So, this approach was forsaken (perhaps prematurely).

4.5.2 Interval triggers

The second attempt at supporting floating-point arithmetic was somehow orthogonal. The goal was no longer to be as clever as Gappa, but rather to support all the exceptional behaviors of SMT-LIB floating-point theory, which had appeared in the meantime [BTRW15]. Mohamed Iguernelala and Sylvain Conchon implemented a new trigger mechanism in Alt-Ergo, Kailiang Ji wrote a bridge between the SMT-LIB format and Why3, and Clément Fumex and I wrote and realized a new floating-point theory for Why3 [CIJ+17]. Contrarily to the previous attempt, we wanted to minimize the changes to Alt-Ergo, hence the reuse of Why3's theories. This led to an architecture based on three layers:

1. The first layer consists of a set of generic axioms about floating-point arithmetic. They make it possible to reason about the shape of floating-point expressions and to reduce floating-point constraints to real ones using a rounding operator, when floating-point numbers are finite.

2. The second layer plays an important role in this new approach, as it enables effective cooperation between floating-point and real reasoning. It is made of two parts: an axiomatic part that provides properties about rounding operators in an extended syntax of Alt-Ergo and an interval-matching mechanism to instantiate these axioms.

3. The third layer is a reasoning engine for nonlinear real arithmetic (NRA) that is able to provide bounds for terms. It is extended to partially handle some operators like rounding, exponentiation, integer logarithm, maximum, and absolute value. More precisely, this consists in enabling calculations when these operators are applied to constants.

There is not much to say about the third layer, as Alt-Ergo's procedure is a rather standard nonlinear arithmetic solver based on interval arithmetic. The only peculiarity is the ability to compute with constants, but that is straightforward to implement. So, let us consider the two other layers.

Layer 1

The first layer axiomatizes floating-point arithmetic in a way that is compatible both with the IEEE-754 standard and the SMT-LIB standard. The main difference between both standards is that, in SMT-LIB, there is only one NaN. These axioms are expressed as a Why3 theory [FMM17]. It starts with a unique type `t`, which will later be instantiated for each of the standard formats. We also need a few predicates to characterize the floating-point values:

```
predicate is_finite    t
predicate is_infinite  t
predicate is_nan       t
```

The Why3 syntax here means that these predicates take a single argument of type `t`. Then two axioms ensure that a floating-point datum is either finite or infinite or NaN. Note that our formalization uses abstract predicates, but for SMT solvers that support enumerations, *e.g.*, Alt-Ergo, one could also imagine a less abstract formalization that does not require these two axioms.

```
axiom is_not_nan: forall x:t.
  (is_finite x \/ is_infinite x) <-> not (is_nan x)
axiom is_not_finite: forall x:t.
  not (is_finite x) <-> (is_infinite x \/ is_nan x)
```

Our formalization does not give direct access to the significand or the exponent of a finite floating-point number. It just provides an abstract function from `t` to the set of a real numbers.

```
function to_real t : real
```

As for formats, the largest representable number, *i.e.*, $\beta^{e_{\max}} \cdot (1 - \beta^{-\varrho})$, is left abstract. A predicate characterizes real numbers in the range of finite numbers.

```
constant max_real : real
predicate in_range (x:real) = -max_real <= x <= max_real
```

Not all the real numbers that are in range are representable, so we arrive to the notion of rounding. Let us first enumerate the five rounding modes supported by the standard:

```
type mode = RNE | RNA | RTP | RTN | RTZ
```

The first two modes round to nearest, with tie breaking to even (RNE) or away from zero (RNA). The last three modes are directed rounding: toward $+\infty$ (RTP), toward $-\infty$ (RTN), and toward zero (RTZ).

For each target format, a rounding operator tells which floating-point number should be chosen to represent a real number. If the real number is representable, then the choice is trivial. (The only peculiarity comes from the existence of signed zeroes, which we will not detail here.) Otherwise, the floating-point number should be the closest one according to the rounding mode. As explained in Section 2.1.1, such a rounding operator assumes that the format has no upper bound on the exponent range. That way, we do not have to bother with the issue of overflow yet.

```
function round mode real : real
```

Note that, if one wanted to use such a concrete definition in an SMT solver, one would also need $\log_\beta$ to be defined. This can be avoided by instead providing a function from $|x|$ to β^e, since this function can be finitely axiomatized using just inequalities for any format, as was done in REALIZER [LMRW14]. In our case, we do not care about these details, as `round` is kept as an abstract function.

Our rounding operator ignores the issue of overflow, but this issue still has to be handled. We do so by defining the following predicate. It will be used to decide whether a rounded value is relevant or not as a result of a floating-point operator.

```
predicate no_overflow (m:mode) (x:real) =
  in_range (round m x)
```

We now have enough definitions to state what the behavior of the floating-point addition is when its inputs are finite numbers.

```
function add mode t t : t
axiom add_finite: forall m:mode, x y:t.
  is_finite x -> is_finite y ->
  no_overflow m (to_real x + to_real y) ->
```

```
  is_finite (add m x y) /\
  to_real (add m x y) = round m (to_real x + to_real y)
```

As for the overflow case, here is a small excerpt of the `add_special` axiom which covers all the exceptional behaviors of the floating-point addition according to the standard:

```
axiom add_special: forall m:mode, x y:t.
  let r = add m x y in
  ... (* 6 other conjuncts *) ... /\
  (is_finite x /\ is_finite y /\
   not no_overflow m (to_real x + to_real y)
   -> same_sign_real r (to_real x + to_real y) /\
      overflow_value m r)
```

The `same_sign_real` relation tells which sign the final result has, assuming the infinitely precise intermediate result is nonzero (which is the case if it overflowed). The `overflow_value` relation then selects the final result depending on the rounding mode and its sign.

An important property of this Why3 formalization is that it is a straightforward translation from the IEEE-754 standard. That does not mean that it is trivially error-free though, since some hypotheses could well have been forgotten or an `r` could have mistyped for an `x`.[11] Thus, this formalization has been realized using the Coq proof assistant. What this means is that concrete definitions were given to all the abstract declarations, using the operators of Section 4.3.2, *e.g.*, the `Bplus` function of Flocq for the floating-point addition. Then, we have proved in Coq that all the axioms hold. This gives a high level of confidence in the correctness of the Why3 formalization.

Layer 2

As the `round` function is left abstract, the above axioms are not sufficient in non-exceptional cases. So, a few more axioms are added to handle rounding operators. For example, propagating a range can be done using the following axiom, where m encompasses both the format and the rounding mode, for simplicity:

$$\forall x\, i\, j\, m,\ i \leq x \leq j \Rightarrow \square_m(i) \leq \square_m(x) \leq \square_m(j). \tag{4.5}$$

Unfortunately, such an axiom cannot be efficiently instantiated using generic E-matching techniques of SMT solvers, as it is not always possible to provide good triggers for them. A *trigger* for an axiom $\forall \vec{x},\ \varphi(\vec{x})$ is a term (or a set of terms) that covers all variables $\vec{x}$. Based on this definition, the triggers inferred for the axiom above would presumably be $\{\square_m(i), \square_m(x), \square_m(j)\}$. This set of terms, however, would make it impossible to apply the axiom when trying to prove the following formula:

$$2 \leq a \leq 4 \implies 2 \leq \circ(a) \leq 4.$$

Indeed, the only way to instantiate the trigger is by choosing $m = \text{RNE}$ and $i = j = x = a$, since the only rounded value in the formula is $\circ(a)$. Yet, the proper way to instantiate the axiom is $i = 2$, $j = 4$, $x = a$. Then, effectively computing $\circ(2)$ and $\circ(4)$ makes it possible to conclude.

[11] This actually happened.

$$
\begin{array}{ll}
\forall x\, i\, j, & i \leq x \leq j \Rightarrow -2^{\alpha} \leq \circ(x) - x \leq 2^{\alpha} \\
& \qquad \text{where } \alpha = \mathtt{ilog_2}(\max(|i|,\, |j|,\, 2^{e_{\min}+\varrho-1})) - \varrho, \\
\forall x\, i\, j\, m, & i \leq x \leq j \Rightarrow \square_m(i) \leq \square_m(x) \leq \square_m(j), \\
\forall x\, i\, m, & \square_m(x) < i \Rightarrow x < \triangle(i), \\
\forall x\, i\, m, & i \leq \square_m(x) \Rightarrow \triangle(i) \leq \square_m(x), \\
\forall x\, m_1\, m_2, & \square_{m_1}(\square_{m_2}(x)) = \square_{m_2}(x), \\
\forall x\, y\, m, & x \leq y \Rightarrow \square_m(x) \leq \square_m(y), \\
\forall x\, y\, m, & \square_m(x) < \square_m(y) \Rightarrow x < \square_m(y), \\
\forall x\, y\, m, & \square_m(x) < \square_m(y) \Rightarrow \square_m(x) < y, \\
\forall x\, y\, m, & |x| \geq 1 \Rightarrow |x| = 2^{\mathtt{ilog_2}(|x|)} \Rightarrow \square_m(x \cdot \square_m(y)) = x \cdot \square_m(y), \\
\forall x\, y\, m, & \square_m(x) < \square_m(y) \Rightarrow \square_m(\square_m(x) - \square_m(y)) < 0, \\
\forall x\, y\, m, & \square_m(x) < -\square_m(y) \Rightarrow \square_m(\square_m(x) + \square_m(y)) < 0.
\end{array}
$$

Figure 4.13 – Overview of the axioms of Layer 2.

In order to efficiently reason modulo our rounding properties, we have annotated them with two kinds of triggers: *syntactic* and *interval* triggers. Syntactic triggers are those described above, which are used by the generic E-matching mechanism. A suitable set of syntactic triggers for Equation 4.5 is the singleton $\{\square_m(x)\}$. Interval triggers are guards used to carefully instantiate the variables that are not covered by syntactic triggers. The purpose of this kind of triggers is twofold. First, it avoids cases where syntactic triggers are either too permissive or too restrictive. Second, it takes the current arithmetic environment into account to guide the instantiation process. A suitable interval trigger for Equation 4.5 is the set $\{i \leq x, x \leq j\}$.

When Alt-Ergo tries to build all the instances of an axiom, it first uses E-matching to satisfy the syntactic triggers, then it queries the NRA engine to satisfy the interval triggers. Figure 4.13 gives an overview of the axioms used in Layer 2. In the first four axioms, the antecedents are interval triggers.

4.5.3 Assessment

As it stands, the proper integration of a mechanism *à la* Gappa into Alt-Ergo has yet to be achieved. Our first attempt was the closest to Gappa's spirit, but the lack of a deep collaboration between the floating-point procedure and the rest of the SMT solver caused performance issues. The second attempt provides a much better collaboration through the use of interval triggers, but it is far from being as powerful as Gappa. At least, it was the occasion to fully support exceptional floating-point values. And interval triggers are a success on their own and are here to stay.

What this tells us is that Gappa should be seen as an instantiation engine for a very specific language. Indeed, an SMT solver is not able to instantiate an equality such as $\tilde{u} \cdot \tilde{v} - u \cdot v = (\tilde{u} - u) \cdot v + u \cdot (\tilde{v} - v) + (\tilde{u} - u) \cdot (\tilde{v} - v)$, as there are no syntactic triggers. Perhaps some new notion of trigger would make it possible to emulate Gappa's behavior in an SMT solver. Interval triggers have solved half the problem, in that quantifiers no longer

need to be instantiated using ground terms only. The missing half is the ability to perform a top-down search rather than a bottom-up search.

Chapter 5

Miscellaneous

Chapter 3 showed how to formalize floating-point arithmetic and interval arithmetic inside Coq and how to use them to automatically and formally verify inequalities on real-valued expressions. Chapter 4 went further into the formalization of floating-point arithmetic and showed how to automatically verify floating-point algorithms using tools such as Gappa, Coq, and Alt-Ergo.

Yet, my research interests on automatic verification do not revolve around real and floating-point expressions only. I have also looked at a few other domains where proofs benefit from automation, such as linear arithmetic and hybrid systems, as shown in Section 5.1.

Conversely, for the most basic blocks of floating-point arithmetic, the algorithms are so specific that an automated verification does not necessarily make sense. In that case, we are down to a manual formalization. Section 5.2 presents my work on the floating-point predecessor and successor, on double rounding and rounding to odd, and on the emulation of the fused multiply-add operator.

Finally, Section 5.3 gives an overview of some larger applications I have worked on, including the verification of a numerical scheme for the wave equation, the disproving of Masser-Gramain's conjecture, and the implementation of a formally verified library for arbitrary-precision integer arithmetic.

5.1 Decision procedures

Section 5.1.1 shows how Alt-Ergo's decision procedure for linear arithmetic was improved by speeding up the detection of unsatisfiability and of implied equalities. Section 5.1.2 then considers an extension of linear arithmetic that uses symbolic powers as coefficients, as they might occur when verifying a library for arbitrary-precision integer arithmetic. Finally, Section 5.1.3 looks at the automatic generation of invariants for hybrid systems.

5.1.1 Linear integer arithmetic

Decision procedures for the quantifier-free linear fragment over integers (QF-LIA) are widely studied in Satisfiability Modulo Theories. Most of the procedures used by state-of-the-art SMT solvers are extensions of either the Simplex algorithm or the Fourier-Motzkin method. Both techniques first relax the initial problem to the rational domain and then proceed by branching/cutting methods or by projection [Sch98]. Given a conjunction of constraints

$\bigwedge_i \sum_j a_{i,j}\, x_j + b_i \leq 0$, usually denoted $A \cdot x + b \leq 0$, the goal is to find an instantiation of the variables x_j with integers satisfying these constraints or a contradiction if they are unsatisfiable.

The Simplex algorithm is meant to solve this problem over rational numbers (QF-LRA) rather than integers. There are three well-known ways of extending this method to decision procedures over integers: *branch-and-bound*, *Gomory's cutting-planes*, and *branch-and-cut*. Roughly speaking, these extensions prune non-integer solutions from the search space until they find an integer assignment or a contradiction.

The Fourier-Motzkin algorithm is another method to solve the problem over rational numbers. It performs successive variable eliminations in a breadth-first manner generating additional constraints with fewer variables. The original system is satisfiable over rational numbers if and only if a fixpoint is reached without deriving a trivially inconsistent inequality $c \leq 0$ where c is a positive rational number. The *Omega-Test* extends this algorithm to a decision procedure over integers by performing additional projection-based checks when the constraints are satisfiable in the rational numbers [Pug91].

The Fourier-Motzkin algorithm has the property of finding implied equalities, which is quite useful for SMT solvers based on the Nelson-Oppen approach. But it does not scale in practice due to the double exponential number of inequalities it generates. By contrast, the Simplex algorithm is simply exponential in the worst case and it behaves rather well in practice, but it does not tell us anything about the implied equalities of the problem. In collaboration with François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, and Alain Mebsout, we devised a novel decision procedure for conjunctions of quantifier-free linear integer arithmetic constraints, which emulates a run of the Fourier-Motzkin algorithm using the Simplex algorithm [BCC+12]. Later, Bromberger and Weidenbach showed that our method was much more general than we originally envisioned [BW16].

Let us consider the polytope defined by $A \cdot x + b \leq 0$, as well as the hypercubes (with edges parallel to the axes) that fit inside it. In other words, we are interested in centers c and radii r such that

$$\forall x \in \mathbb{R}^n,\ \|x - c\|_\infty \leq r \Rightarrow A \cdot x + b \leq 0.$$

These hypercubes are interesting for several reasons. First, if one can find arbitrarily large hypercubes that fit in the polytope, it means that the polytope contains infinitely-many integer points. Second, if one finds a hypercube of radius at least $\frac{1}{2}$ that fits in the polytope, then rounding its center to the nearest point gives a point with integer coordinates that is part of the hypercube and thus of the polytope. Third, if the largest hypercube that fits in the polytope has radius zero, then the polytope is actually a hyperplane, so there is an implied equality. Finally, if the polytope contains no hypercube, then it is empty.

The important thing to notice is that a hypercube fits in the polytope if its center c satisfies $A \cdot c + b + r \cdot \gamma \leq 0$ with $\gamma_i = \|(a_{i,j})_j\|_1$. This means that all the properties above can be decided at once by running the Simplex algorithm to find a vector c and a radius r that maximize r subject to $A \cdot c + b + r \cdot \gamma \leq 0$.

When the polytope is a hyperplane, this does not tell us which one of the problem inequalities happens to be an equality. Moreover, if one of the inequalities happens to have a lower bound (in addition to the upper bound 0), we would like to know about it. To do so, we can consider some kind of dual linear problem. We want to find a vector λ that maximizes $b^\top \cdot \lambda$ subject to $A^\top \cdot \lambda = 0$ and $\lambda \geq 0$ and $\sum \lambda_i > 0$. (Bromberger and Weidenbach turned the constraint $\sum \lambda_i > 0$ into $\gamma^\top \cdot \lambda = 2$.)

This vector λ describes one of the linear combinations that the Fourier-Motzkin algorithm would compute, since $A^\top \cdot \lambda = 0$ ensures that the combination has a constant value, while $\lambda \geq 0$ and $\sum \lambda_i > 0$ ensure the signs of the coefficients are adequate. Moreover, since we are maximizing $b^\top \cdot \lambda$, we get a positive constant if there is one, which means that the original problem is unsatisfiable. Otherwise, for any i such that $\lambda_i \neq 0$, we can deduce

$$\frac{b^\top \cdot \lambda}{\lambda_i} \leq \sum_j a_{i,j}\, x_j \leq 0.$$

The main difference with the Fourier-Motzkin algorithm (not counting the much lower time complexity) is that our approach might find only one implied equality or lower bound rather than all of them. It means that, in order to progress further, one variable has to be eliminated at this point, *e.g.*, by enumerating all the possible cases implied by the newly found lower bound. This work was implemented in the Alt-Ergo SMT solver.

5.1.2 Pseudo-linear arithmetic

In some cases, the system of inequalities might look like it fits into the linear fragment, but some coefficients are not constant. This makes SMT solvers unusable and thus requires a large proof effort from the user. Raphaël Rieu-Helft encountered this issue when verifying some algorithms from the GNU Multi-Precision library and we had to devise a new approach to solve such systems [MRH18].

The GNU Multi-Precision library,[1] GMP for short, is a widely used library for arithmetic on integers and rational numbers of arbitrary size. The lower layer of GMP is the `mpn` component, which is dedicated to nonnegative integers and is used by all the other components.

Consider the `mpn_add_n` function, which adds two integers x and y in order to produce an integer r. More details on the actual representation is given in Section 5.3.3. What matters here is that an `mpn` integer a is a sum $\sum_k a_k \beta^k$ with the radix β being 2^{32} or 2^{64}, depending on the architecture. The addition algorithm iteratively adds the β-digits x_k and y_k from the least significant one to the most significant one, propagating a carry c along the way. Figure 5.1 shows the Why3 task that corresponds to the proof of preservation of the following loop invariant for i ranging from 0 to the size of x and y:

$$\sum_{0 \leq k < i} x_k \beta^k + \sum_{0 \leq k < i} y_k \beta^k = c\beta^i + \sum_{0 \leq k < i} r_k \beta^k.$$

Partial sum are represented using the function `value`. Hypothesis `H0` expresses the invariant before computing a β-digit, with `r1` (resp., `c1`) representing the current sum (resp., the carry). Hypotheses `H3` and `H4` state that the limb of x (resp., y) at position i is `lx` (resp., `ly`). Hypothesis `H1` explains how `res` and `c` are computed from `lx`, `ly`, and `cy1`. The remaining hypotheses glue `r`, `r1`, and `res` together.

Notice that the linear combination $\texttt{H5} - \texttt{H4} - \texttt{H3} + \texttt{H2} + \beta^i \cdot \texttt{H1} + \texttt{H0}$ simplifies to an equality equivalent to `G`. So, proving that the goal is a consequence of the hypotheses seems trivial enough. Unfortunately, since β^i and β^{i+1} are not constant, this verification condition falls outside the QF-LIA fragment. Thus, the user ends up doing the work by hand, adding numerous cuts until the SMT solvers are satisfied. And that is for one of the simplest algorithms. It gets worse for more complicated algorithms such as division.

[1] `http://gmplib.org/`

```
axiom H0: value r1 i + power radix i * c1 = value x i + value y i
axiom H1: res + radix * c = lx + ly + c1
axiom H2: value r i = value r1 i
axiom H3: value x (i+1) = value x i + power radix i * lx
axiom H4: value y (i+1) = value y i + power radix i * ly
axiom H5: value r (i+1) = value r i + power radix i * res
goal G:   value r (i+1) + power radix (i+1) * c
        = value x (i+1) + value y (i+1)
```

Figure 5.1 – One verification condition for the addition algorithm.

Reflection-based verification

To recover a semblance of automation, a solution is to resort to reflection as explained in Section 2.3.2. We need a small WhyML algorithm to prove that a linear equality is the linear combination of other linear equalities, except that the coefficients involve powers of β.

For now, let us ignore the intricacy due to the coefficients and let us focus on the linear solver. We represent the hypotheses and goal by a matrix. The decision procedure, partly shown in Figure 5.2, performs a Gaussian elimination (function `gauss_jordan`). In case of success, we obtain a vector of coefficients and we check whether the corresponding linear combination of the context is equal to the goal (function `check_combination`). Otherwise, the algorithm returns `False` and proves nothing, since its postcondition has `result = True` as antecedent.

As is done in Coq with the tactics `lia` and `lra`, this is a proof by certificate [Bes07]. Indeed, we check if the linear combination of the context returned by `gauss_jordan` matches the goal. There is no need to prove the Gaussian elimination algorithm itself, nor to define a semantics for the matrix passed to it as a parameter. In fact, we do not prove anything about the content of any matrix in the program. This makes the proof of the decision procedure very easy in relation to its length and intricacy.

Let us now examine the contract of the decision procedure. The postcondition states that the goal holds if the procedure returns `True`, for any valuations `y` and `z` of the variables such that the hypothesis equalities hold (function `interp_ctx`). The `raises` clause expresses that an exception may escape the procedure (typically an arithmetic error, as we allow the field operations to be partial). In that case, nothing is proved.

Let us look at the coefficients of our decision procedure now. They are represented as pairs of a rational number and a (symbolic) power of β. Addition, multiplication, and multiplicative inverse, are defined over these coefficients. Addition is partial, since one may only add two coefficients with equal exponents. If this is not the case, the addition raises an exception, which is accounted for in the specification of the decision procedure (exception `C.Unknown` in Figure 5.2). Note that exponents do not have to be structurally equal, only to have equal `interp_exp` interpretations for all values of `y`, which can be automatically proved within the decision procedure.

Since the operations on coefficients are partial, the procedure is necessarily incomplete. Nonetheless, it was able to prove all the linear goals we encountered while verifying GMP's

```
type expr = Term coeff int | Add expr expr | Cst coeff
type equality = (expr, expr)

(* these functions live in the logical world;
    they are only meant for the specification of the procedure *)

function interp_eq (g: equality) (y: vars) (z: C.cvars): bool
= match g with (g1, g2) -> interp g1 y z = interp g2 y z end

function interp_ctx (l: list equality) (g: equality)
  (y: vars) (z: C.cvars): bool
= match l with
  | Nil -> interp_eq g y z (* goal *)
  | Cons h t -> (interp_eq h y z) -> (interp_ctx t g y z)
  end

(* this is the actual decision procedure and it calls effectful functions *)

let linear_decision (l: list equality) (g: equality) : bool
  ensures { forall y z. result = True -> interp_ctx l g y z }
  raises  { C.Unknown -> true }
= ...
  fill_ctx l 0;
  let (ex, d) = norm_eq g in
  fill_goal ex;
  let ab = m_append a b in
  let cd = v_append v d in
  match gauss_jordan (m_append (transpose ab) cd) with
  | Some r -> check_combination l g (to_list r 0 ll)
  | None -> False
  end
```

Figure 5.2 – Decision procedure for linear equation systems.

algorithms. Note that, while the procedure is specialized for GMP goals, reasoning is done almost entirely in the generic linear decision procedure `linear_decision`, which does not rely on any specific property of the `coeff` type and can be instantiated with other types.

Reification and interpretation

While we have a verified WhyML procedure meant to prove systems of linear equalities, we have not yet explained how we actually use it. Indeed, Why3 is meant to verify programs, nothing more. So, we have extended Why3 with reflection capabilities, in order to be able to use verified programs to verify subsequent programs [MRH18].

As shown in Figure 5.2, we have written a procedure, which we want to apply to a goal such as the one on Figure 5.1. The specification of the procedure tells that we can assume the following proposition:

```
forall y z. result = True -> interp_ctx l g y z
```

But there are two issues. First, what is `result`? In Section 2.3.2, it was an expression in the logical world, so it was readily available. Here, it denotes the result of a WhyML program. The second issue is that the goal we want to prove is a system of equalities while the postcondition of `linear_decision` starts with `interp_ctx`. More precisely, the postcondition of `linear_decision` might be the goal we want to prove, but only if we are able to find suitable values for the arguments l and g of the procedure and for the quantifiers y and z of the postcondition.

The second issue is well-known and it has been solved over and over in the context of reflection, be it computational or not. The approach Raphaël Rieu-Helft and I have implemented in Why3 to perform reification is based on inverting interpretation functions on the fly to perform the reification. These functions are not marked in any specific way; Why3 just assumes that any function symbol causing a discrepancy between the postcondition and the goal can be used to interpret a reified object.

Here, Why3 tries to match `(interp_ctx l g y z)` with the conclusion of the goal. This fails, so Why3 assumes that `interp_ctx` is an interpretation function and it unfolds it. Since the function matches on a list, Why3 assumes that the `Nil` case is dedicated to the conclusion, so it tries to match `(interp_eq g y z)` (see Figure 5.2) with the conclusion. Again, there is a mismatch, so Why3 assumes that `interp_eq` is an interpretation function, too. This time, it partly matches, since there is an equality in both cases. So, Why3 continues its reification by first matching the left-hand sides together and then the right-hand sides.

The main difference with traditional approaches is that we do not offer any meta-language for the users to write their reification procedure. We just provide various heuristics to build the reified objects by looking at the interpretation functions [MRH18]. This is similar to the approach followed by the `quote` tactic of Coq, but in a more expressive way. Note that the reification process does not modify the trusted code base of Why3, since the standard rewriting engine of Why3 is used to turn the reified objects back into a proposition that looks like the goal. In case of bug, the corresponding proposition holds nonetheless (assuming the user verified `linear_decision`), but it does not help in proving the goal.

We still have an issue. We are trying to use the postcondition of a WhyML program to verify a goal and this postcondition mentions the value computed by this imperative program when fed the result of the reification process. Due to their side effects, functions from WhyML programs only have abstract declarations in the logical world (as opposed to the concrete

logical functions used in Section 2.3.2). Therefore, they cannot be unfolded by automatic provers or by Why3's rewriting engine.

In order to compute the results of decision procedures such as the previous one, we have added an interpreter to Why3. It operates on an ML-like intermediate language that corresponds to WhyML programs from which logical terms, assertions, and ghost code, were erased, thereby assuming that the program was proved beforehand and that the preconditions are met. This intermediate code is produced by the extraction mechanism of Why3, which is used to produce OCaml and C programs from proved WhyML programs (see also Section 5.3.3). Our interpreter provides built-in implementations for some axiomatized parts of the Why3 standard library, such as integer arithmetic and arrays. This time, the trusted code base of Why3 is enlarged, but not much, since we are piggybacking on the existing extraction mechanism, which makes the interpreter straightforward.

5.1.3 Invariant generation for hybrid automata

When verifying a program with loops (or recursive functions), one of the most difficult parts lies in finding the loop invariants, from which the correctness of the code can be deduced. That is what makes *abstract interpretation* so alluring, as it automates the discovery of invariants of a given shape [CC77].

Daisuke Ishii and I investigated the search of invariants in the specific case of *hybrid systems* [IMN13]. Hybrid systems are transition systems with continuous dynamics. They are a good model for embedded systems that have a strong coupling with the physical environment.

Hybrid automaton

Definition 5.1 *A* hybrid automaton *is a tuple* $\langle L, V, \mathit{Init}, G, R, F, I\rangle$ *that consists of the following components:*

- *a finite set* $L = \{\ell_1, \ldots, \ell_p\}$ *of* locations*,*
- *a finite set* $V = \{v_1, \ldots, v_q\}$ *of real-valued* variables*,*
- *a set of* initial states $\mathit{Init} \subseteq L \times \mathbb{R}^V$*,*
- *a family* G *of* guard conditions $g_{\ell,\ell'} \in \mathbb{R}^V$ *for* $\ell, \ell' \in L$*,*
- *a family* R *of* reset functions $r_{\ell,\ell'} : \mathbb{R}^V \to \mathbb{R}^V$ *for* $\ell, \ell' \in L$*,*
- *a family* F *of* vector fields $f_\ell : \mathbb{R}^V \to \mathbb{R}^V$ *for* $\ell \in L$*,*
- *a family* I *of* location invariants $i_\ell \in \mathbb{R}^V$ *for* $\ell \in L$*.*

A (finite or infinite) *execution* is a sequence of states $\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots$, with $\sigma_i \in L \times \mathbb{R}^V$ and $\sigma_0 \in \mathit{Init}$. An arrow $\xrightarrow{*}$ is either a *continuous evolution* phase $\xrightarrow{t}$ where $t > 0$ or a *discrete transition* phase $\xrightarrow{0}$. Its operational semantics is given by the following rules, where $\dot{\Phi}$ denotes the differential of Φ with respect to time.

$$\frac{t > 0 \qquad \nu_0 = \Phi(0) \qquad \nu_t = \Phi(t) \qquad \forall t' \in [0; t],\ \dot{\Phi}(t') = f_\ell(\Phi(t')) \wedge \Phi(t') \in i_\ell}{\langle \ell, \nu_0\rangle \xrightarrow{t} \langle \ell, \nu_t\rangle},$$

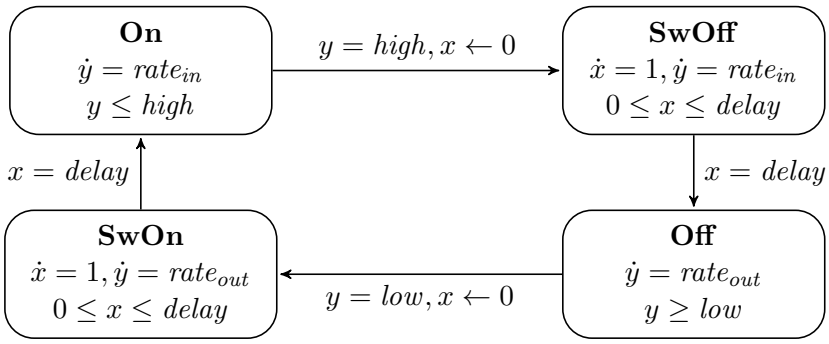


Figure 5.3 – Water-level monitor.

$$\frac{\nu \in g_{\ell,\ell'} \qquad \nu' = r_{\ell,\ell'}(\nu) \qquad \nu' \in i_{\ell'}}{\langle \ell, \nu \rangle \xrightarrow{0} \langle \ell', \nu' \rangle}.$$

Note that, in this model, executions are not deterministic. For example, discrete transitions do not necessarily happen as soon as $\Phi(t')$ hits a set $g_{\ell,\ell'}$. They might be delayed as long as the invariant $\Phi(t') \in i_\ell$ is satisfied.

Figure 5.3 shows an example of a deterministic hybrid automaton. The nodes represent continuous evolutions; they display the name of the state, the differential equations, and the invariant. The arrows represent discrete transitions; they display the guard condition and the reset function, if any. This hybrid automaton models a controlled water tank. In location Off, it supplies water at a constant rate $rate_{out}$, so the water level y decreases as $\dot{y}(t) = rate_{out}$. In location On, the system pumps water to refill the tank, which results in the water level changing as $\dot{y}(t) = rate_{in}$. A sensor observes y and switches between the locations On and Off when the level reaches the thresholds *low* or *high*. However, it takes *delay* seconds for switching, hence the additional locations SwOn and SwOff. This delay is tracked by the variable x, which is reset to zero before entering the locations SwOn and SwOff.

We are interested in *safety properties* (or *inductive invariance*) satisfied by a hybrid automaton. Such a property P holds for $\sigma_0 \in \mathit{Init}$ and is preserved along any execution path starting from σ_0. Our goal is to verify that a given property is a safety property for a given hybrid automaton. In the example of the water-level monitor, we might be interested in checking that the tank is never too empty nor does it exceed its capacity, *i.e.*, $min \leq y \leq max$.

Imperative program

Deductive program verification led to various methods to verify the safety of programs, so let us reuse them to analyze hybrid automata. The first step is to define a minimal imperative language with non-deterministic semantics. The syntax is as follows.

$$s ::= \texttt{skip} \mid s\texttt{;}s \mid \texttt{evolve}\ t \mid \texttt{trans}.$$

The *program state* (denoted S or S_i) is a map from variable names to program values. A special variable $\sigma = \langle \ell, \nu \rangle$ is associated to the "current" state ($\in L \times \mathbb{R}^V$) of the hybrid-automaton execution. We denote $[\![e]\!]_S$ the term obtained by replacing each free variable of an expression e by its associated value in the program state S. We denote $S[x \mapsto v]$ the program state obtained by modifying S so that variable x has value v. The operational semantics of

the language is defined by the following rules. The rules for `skip` and sequence are the usual ones.

$$\frac{}{S, (\texttt{skip}; s) \rightsquigarrow S, s} \qquad \frac{S_1, s_1 \rightsquigarrow S_2, s_2}{S_1, (s_1; s_3) \rightsquigarrow S_2, (s_2; s_3)} \qquad \frac{}{S, \texttt{evolve } 0 \rightsquigarrow S, \texttt{skip}}$$

$$\frac{[\![\sigma]\!]_S \xrightarrow{t} \sigma'}{S, \texttt{evolve } t \rightsquigarrow S[\sigma \mapsto \sigma'], \texttt{skip}} \qquad \frac{[\![\sigma]\!]_S \xrightarrow{0} \sigma'}{S, \texttt{trans} \rightsquigarrow S[\sigma \mapsto \sigma'], \texttt{skip}}$$

We can now express any partial execution of a hybrid automaton as a non-blocking program, *i.e.*, a program that reduces to `skip`. Therefore, we can state the safety property of the automaton as a property that every non-blocking program must satisfy in its final state [IMN13].

Lemma 5.1 (Safety) *If, for all non-blocking programs run from an initial program state $\sigma_0 \in Init$, property P holds in the final program state, then P is a safety property for the hybrid automaton (up to the first Zeno point,[2] if any).*

Notice that `evolve` 0 is allowed, which means that any execution can be represented by a program in a canonical form that alternates `evolve` and `trans` commands.

Let us define a logical calculus over these programs. Since we are not dealing with reachability but only safety, weakest preconditions and strongest postconditions are dual from each other for our purpose. We chose the latter so as to follow the direction of time.

$$\mathrm{SP}(P, \texttt{skip}) \stackrel{\text{def}}{=} P, \qquad \mathrm{SP}(P, s_1; s_2) \stackrel{\text{def}}{=} \mathrm{SP}(\mathrm{SP}(P, s_1), s_2),$$

$$\mathrm{SP}(P, \texttt{evolve } t) \stackrel{\text{def}}{=} \exists \Phi,\ P[\nu \leftarrow \Phi(0)] \wedge \Phi(t) = \nu \wedge (\forall t' \in [0, t],\ \dot{\Phi}(t') = f_\ell \wedge \Phi(t') \in i_\ell),$$

$$\mathrm{SP}(P, \texttt{trans}) \stackrel{\text{def}}{=} \exists \ell', \nu',\ P[\ell \leftarrow \ell', \nu \leftarrow \nu'] \wedge \nu' \in g_{\ell',\ell} \wedge \nu = r_{\ell',\ell}(\nu') \wedge \nu \in i_\ell.$$

Lemma 5.2 (Soundness of SP**)** *For any program s, if the initial state satisfies a given property P, the final state satisfies* $\mathrm{SP}(P, s)$*, assuming the program terminates.*

As we will later pass the verification conditions to automated tools, it is important to eliminate as many quantifiers as possible beforehand. For example, $\mathrm{SP}(P, \texttt{trans})$ has the form $\exists \ell',\ Q[\ell']$. This is equivalent to the disjunction $Q[\ell_1] \vee \ldots \vee Q[\ell_p]$ with $\ell_1, \ldots, \ell_p$ all the locations. In the case of $\mathrm{SP}(P, \texttt{evolve } t)$, one can get rid of $\exists \Phi$ if the ordinary differential equation admits a closed form.

Inductive verification

The statement of Lemma 5.1 is cumbersome, as it requires verifying infinitely-many programs. We can build weaker yet more practical variants of it, by only considering a bounded number of programs. The approach is as follows. Let us assume that there is a predicate P^+ such that $P^+ \Rightarrow P$ and

- from an initial state, any execution reaches a state satisfying P^+ after alternating at most m continuous evolutions and m discrete transitions,

[2] A hybrid system might alternate infinitely-many continuous evolutions and discrete transitions in a finite time [ZJLS01]. The accumulation point is a Zeno point, named from the philosopher's paradox.

- from any state satisfying P^+, any execution reaches a state satisfying P^+ after alternating at most n continuous evolutions and n discrete transitions.

Therefore, if we can exhibit some lengths m and n and some predicate P^+, verifying the safety property becomes simple. Here is the case $m = 0$ and $n = 1$. See the article for the general case [IMN13].

Theorem 5.3 (Simple case) *If the following formulas hold,*

$$\sigma \in \mathit{Init} \Rightarrow P^+, \tag{5.1}$$

$$\forall t \geq 0,\ \mathrm{SP}(P^+, \texttt{evolve}\ t) \Rightarrow P, \tag{5.2}$$

$$\forall t \geq 0,\ \mathrm{SP}(P^+, \texttt{evolve}\ t\texttt{;trans}) \Rightarrow P^+, \tag{5.3}$$

then P is a safety property.

The algorithm to find P^+ starts from P and adds conjuncts to it to strengthen it. If, at some point, Equation (5.1) no longer holds, then the algorithm gives up (and tries different values of m and n). If both Equations (5.2) and (5.3) holds, the algorithm is done, as Theorem 5.3 can be applied. Otherwise, the algorithm tries to add a conjunct. This conjunct Q is obtained by performing a quantifier elimination on the equation that does not hold. For example, for Equation (5.2), we have

$$Q \stackrel{\text{def}}{=} \mathrm{QE}(\forall \sigma,\ \forall t \geq 0,\ \mathrm{SP}(P^+ \wedge \sigma_0 = \sigma, \texttt{evolve}\ t) \Rightarrow P)[\sigma_0 \leftarrow \sigma].$$

The formula computed for Q is often a large disjunctive formula that is unusable as a loop invariant. For instance, some sub-formulas of Q describe states that are never accepted by the hybrid automaton. Such sub-formulas are not only useless but make the verification process expensive. So, we strengthen Q according to the following strategies:

- *Lemma separation.* We split Q at the topmost disjunction operators and employ one (or several) of the resulting sub-formulas.
- *Location disabling.* When we remove a sub-formula of Q that is related to some location ℓ', we insert the constraint $\ell \neq \ell'$.

Consider the example of Figure 5.3. The initial state is location On and $y = \mathit{low}$. We assume that the constant values satisfy the following inequalities:

$$\mathit{min} \leq \mathit{low} \wedge \mathit{high} \leq \mathit{max} \wedge \mathit{low} < \mathit{high} \wedge 0 < \mathit{delay} \wedge$$
$$\mathit{min} \leq \mathit{low} + \mathit{rate}_{\mathit{out}} \cdot \mathit{delay} \wedge \mathit{high} + \mathit{rate}_{\mathit{in}} \cdot \mathit{delay} \leq \mathit{max}.$$

Our goal is to prove that $P \stackrel{\text{def}}{=} \mathit{min} \leq y \leq \mathit{max}$ is a safety property. If we use both lemma separation and location disabling, the algorithm succeeds at $m = 0$ and $n = 1$ by using

$$P^+ \stackrel{\text{def}}{=} P \wedge \begin{cases} \ell = \mathrm{SwOff} \Rightarrow \mathit{min} + \mathit{rate}_{\mathit{in}} \cdot x \leq y + \mathit{rate}_{\mathit{in}} \cdot \mathit{delay} \leq \mathit{max} + \mathit{rate}_{\mathit{in}} \cdot x, \\ \ell = \mathrm{SwOn} \Rightarrow \mathit{min} + \mathit{rate}_{\mathit{out}} \cdot x \leq y + \mathit{rate}_{\mathit{out}} \cdot \mathit{delay} \leq \mathit{max} + \mathit{rate}_{\mathit{out}} \cdot x. \end{cases}$$

If we use only location disabling, the algorithm succeeds at $m = 0$ and $n = 2$ by using a much simpler invariant:

$$P^+ \stackrel{\text{def}}{=} P \wedge \ell \neq \mathrm{SwOff} \wedge \ell \neq \mathrm{SwOn}.$$

Indeed, P trivially holds in the locations On and Off, but it does not in the locations SwOn and SwOff, as we do not know anything about y there. With $n = 2$ and $\ell \notin \{\mathrm{SwOn}, \mathrm{SwOff}\}$, we can make use of the guard conditions leading to the locations SwOn and SwOff to verify that P holds there.

Assessment

Unfortunately, while the approach of bridging the gap between hybrid system and program verification seemed promising, Daisuke Ishii and I did not get to research it any further, once his visiting time came to an end. So, there are still several questions left unanswered. First, both heuristics, lemma separation and location disabling, are still mostly human guided. We did not get to fully automate it at the time. Second, lemma separation heavily depends on the existence of closed solutions. It would be interesting to study how it could be adapted to approximate approaches like interval arithmetic. Third, our approach is intrinsically limited to hybrid automaton that do not have any Zeno point, as the corresponding imperative program will not execute past it.

Also, our approach presents various similarities to existing ones and we never got to study in depth how they are related. First, loop unrolling and state partitioning are not unlike k-induction [SSS00]. Second, P^+ can be see as an abstraction of the hybrid automaton, which would make its refinement using quantifier elimination similar to the CEGAR approach, *i.e.*, counterexample-guided abstraction refinement [CGJ$^+$03].

5.2 Floating-point arithmetic

Sections 3.2 and 4.3.2 have mostly presented the Flocq library from the point of view of computable floating-point arithmetic operators. But Flocq and its predecessor Pff also describe various floating-point algorithms and formalize their correctness proofs. This section gives an overview of the works I contributed to. Most of them were done in collaboration with Sylvie Boldo [BDKM06, BM08, BM13, BM17].

Section 5.2.1 describes an implementation of the floating-point predecessor and successor, using only arithmetic operations. Section 5.2.2 then presents some properties of the double rounding phenomenon, *i.e.*, rounding first to an extended precision and then to the target precision. It also presents a specific rounding operator, rounding to odd, which alleviates some issues of double rounding. Finally, Section 5.2.3 shows how rounding to odd makes it possible to emulate the fused multiply-add operator.

5.2.1 Predecessor and successor

Section 2.2.2 described an implementation of interval operations using directed rounding. Unfortunately, numerous mainstream processors do not encode rounding direction in floating-point opcodes but rather as a global setting. As a consequence, changing the rounding mode usually has a large impact on performance, as it might require a pipeline flush. So, it would be better if all the interval operations could be performed using the default rounding mode: rounding to nearest. For example, using the floating-point predecessor and successor functions, the following implementation of interval addition, though not optimal, would be sufficient for most use cases:

$$\mathbf{u} + \mathbf{v} \subseteq [\mathrm{pred}(\circ(\underline{u} + \underline{v})); \mathrm{succ}(\circ(\overline{u} + \overline{v}))].$$

So, it is a matter of devising efficient implementations of pred and succ. Thanks to the layout of the IEEE-754 binary interchange format, the ordering of positive floating-point numbers usually matches the ordering of the underlying integer representation. Therefore, predecessor and successor can be implemented by simply adding or subtracting one (depending

on the sign) to the integer representation of their input. One just has to be careful with the extremal cases: signed zeroes, infinities, and NaNs. This approach works well, but some processors cannot efficiently transfer values between floating-point and integer registers.

This begs the question: how to compute the predecessor and successor using only the basic floating-point arithmetic operators? In reality, we do not even care about the exact result of these functions. They can be a bit wrong, as long as we obtain the containment property in the end. Siegfried Rump and Paul Zimmermann proposed some algorithms to this effect, which Sylvie Boldo and I verified using Coq [RZBM09]. Given a finite floating-point number x, the goal is to compute some floating-point number e as small as possible, such that $\circ(x - e) < x < \circ(x + e)$. The format is part of the FLT family, so subnormal numbers are supported.

Note that, for $x = \beta^k$ far from the subnormal range, we have $\mathrm{succ}(x) = x \cdot (1 + \beta^{\varrho-1})$. This means that we cannot choose e smaller than $x \cdot \beta^{\varrho-1}/2$, otherwise we would get $\circ(x + e) = x \neq \mathrm{succ}(x)$. This lower bound on e means that $\circ(x - e)$ is a rather poor approximation of $\mathrm{pred}(x) = x \cdot (1 - \beta^{\varrho})$ for large radices, though. Indeed, the distance between $\circ(x - e)$ and x is about $\beta/2$ units in the last place. For $\beta = 2$, this approach works fine, though.

The first proposed algorithm is as follows. We assume that the format is reasonable. Let us denote ε the floating-point successor of $\beta^{\varrho-1}/2$, and $\eta = \beta^{e_{\min}}$ the smallest positive floating-point number (hence subnormal). We compute e as follows:

$$e = \circ[\varepsilon \cdot |x| + \eta].$$

The correctness proof is a bit tedious, but here is the basic idea. For x in the subnormal range, we have $e \geq \eta$, which is the distance between two subnormal floating-point numbers. For x in the normal range, we have $e \geq \varepsilon \cdot |x|$, so the relative error between x and $x + e$ is too large for $\circ(x + e) = x$ to hold. By monotony, this forces $\circ(x + e) \geq \mathrm{succ}(x)$.

One can also wonder how far we are from the predecessor and successor for $\beta = 2$. We proved that out approach is almost optimal. More precisely, it is optimal for $|x| \notin [\frac{1}{2}; 2] \cdot 2^{e_{\min}+\varrho}$. In that small range comprised of the smallest normal numbers, however, e is a bit overestimated. The proof is ugly, and if it had not be formally verified, it would be difficult to trust.

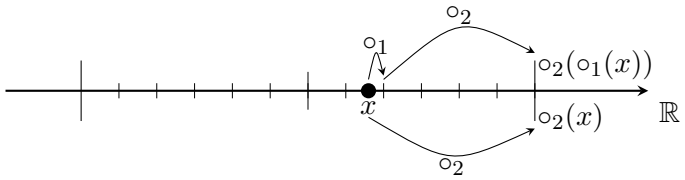
This first algorithm is easy enough to implement, but its speed is disappointing. Again, the issue lies in the design of processors. Most of them tend to be slower when computing with subnormal numbers, if not considerably slower (software emulation). Since η is subnormal, we are paying this overhead every time.

To improve the algorithm, we put a conditional branch, so that subnormal computations can be avoided most of the time. For $|x| \geq 2^{e_{\min}+\varrho+1}$, we use $e = \circ(\varepsilon \cdot |x|)$, which is optimal and fast. Otherwise, we can apply the bit-fiddling algorithm, which is also optimal, and not slower than computing with subnormal numbers.

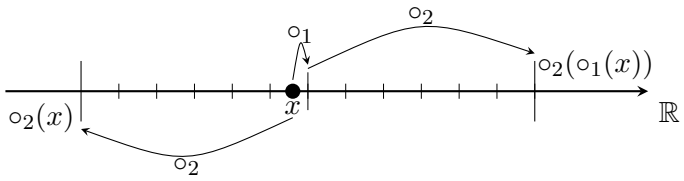
For the sake of completeness, we went a bit further. Indeed, we also proposed a way to compute an optimal value of e in the range $[\frac{1}{2}; 2] \cdot 2^{e_{\min}+\varrho}$ using floating-point arithmetic [RZBM09]. Obviously, that value requires some subnormal computations, so it is only interesting from a theoretical point of view.

5.2.2 Double rounding

Let us now consider a different matter: correct rounding. The IEEE-754 standard requires each arithmetic operation to be performed "as if it first produced an intermediate result



(a) innocuous case



(b) problematic case

Figure 5.4 – Double rounding when rounding to nearest, for an even radix. The two large ticks are numbers in the target format. The medium tick is the midpoint between the two numbers in the working format; it belongs to the extended format. The small ticks represent the other numbers of the extended format.

correct to infinite precision and with unbounded range, and then rounded that result" [IEE08, §4.3]. But what if the intermediate result is produced with an extended but not infinite precision?

Let us denote $\square_2$ and $\circ_2$ some rounding operators to the working format $\mathbb{F}_2$, and $\square_1$ and $\circ_1$ some rounding operators to a format $\mathbb{F}_1$ with a higher precision. Figure 5.4 shows the issue when rounding twice to nearest, for an even radix. In most cases, we have $\circ_2(\circ_1(x)) = \circ_2(x)$. But if $\circ_1(x)$ is sufficiently close to the midpoint of two numbers of $\mathbb{F}_2$ and on the odd side (when breaking ties to even), then we have $\circ_2(\circ_1(x)) \neq \circ_2(x)$. This is called a *double rounding* and we want to avoid it.

Occurrences of double rounding

While this might seem like a far-fetched problem, it occurs surprisingly often in practice. For example, given a floating-point unit that provides only *binary64* operations, can it be used to perform *binary32* operations? In other words, do we have $\circ_{b32}(\circ_{b64}(x \diamond y)) = \circ_{b32}(x \diamond y)$ for x and y two *binary32* numbers? Figueroa showed that it was true for addition, multiplication, division, and square root [Fig95]. With Sylvie Boldo's and my help, Pierre Roux later generalized this result and formalized it in Flocq [Rou14].

A similar example is the set of heterogeneous operations mandated by the IEEE-754 standard [IEE08, §5.4]. These operations have signatures such as $\mathbb{F}_{b64} \to \mathbb{F}_{b32} \to \mathbb{F}_{b32}$. Can these operations be implemented using a *binary64* unit? This time, it is easy to find counter-examples $x, y \in \mathbb{F}_{b64}$ such that $\circ_{b32}(\circ_{b64}(x \diamond y)) \neq \circ_{b32}(x \diamond y)$ [MBdD$^+$18, §7.8.1].

Yet another similar example is the conversion from integers to floating-point numbers. Some processors, *e.g.*, PowerPC, are only able to convert from a 64-bit integer to a *binary64* number. Can we use them to convert to a *binary32* number? This question arose while

developing the CompCert C compiler [BJLM15]. Again, one can find some counterexamples $x \in [0; 2^{63}) \cap \mathbb{Z}$ such that $\circ_{b32}(\circ_{b64}(x)) \neq \circ_{b32}(x)$.

A final example of explicit double rounding is the conversion from binary to decimal formats, where a double rounding phenomenon might again perturb the result [Gol91].

But this situation can also occur unbeknownst of the user. Indeed, as hinted in Section 4.3.1, compilers are allowed to perform some computations in an extended format and then to store them with the format mandated by the user. This can lead to unexpected double rounding [Mon08].

Érik Martin-Dorel, Jean-Michel Muller, and I, studied the effect of double rounding on various well-known algorithms: Fast2Sum, TwoSum, Veltkamp's splitting, some summation algorithms. We also formally verified these results for Fast2Sum and TwoSum [MDMM13].

Rounding to odd

A simple solution to make double rounding innocuous is to use a different rounding operator for the inner rounding: *rounding to odd*. When a real number is not representable, it will be rounded to the adjacent floating-point number with an odd integer significand.

Von Neumann was considering this rounding when designing the arithmetic unit of the EDVAC [vN45, §9.4]. Goldberg later used this rounding when converting binary floating-point numbers to decimal representations [Gol91]. It has also been studied in the context of multistep gradual rounding [Lee89]. Rounding to odd was never more than an implementation detail though, since two extra bits had to be stored in the floating-point registers. It was part of some hardware recipes that were claimed to give a correct result. Sylvie Boldo and I followed a more systematic approach, as we wanted to study it in its own right [BM05].

Let us assume that the radix is even and that the formats $\mathcal{F}_1$ and $\mathcal{F}_2$ are described by the exponent functions φ_1 and φ_2 respectively (see Section 2.1.2). The core result is the following one. If the extended format has two more digits of precision, then using rounding to odd for the inner rounding ensures that double rounding is innocuous.

Theorem 5.4 (Innocuous double rounding) *If the formats satisfy*

$$\forall e \in \mathbb{Z},\ \varphi_1(e) \leq \varphi_2(e) - 2,$$

then

$$\forall x \in \mathbb{R},\ \circ_2\left(\square_1^{\mathrm{odd}}(x)\right) = \circ_2(x).$$

This theorem was especially useful when verifying the conversion between integer and floating-point formats when implementing the CompCert compiler [BJLM15].

One way to emulate rounding to odd is to round toward zero instead. If the inexact exception is raised by the floating-point unit, we set the lower bit of the significand of the result. This forces it to an odd integer, if it was not already so.

5.2.3 Emulation of FMA

Sylvie Boldo and I considered rounding to odd as a way to implement various basic blocks. One such block is the correctly rounded addition of three floating-point numbers. An immediate application is an effective way to emulate a fused multiply-add operator [BM08]. Given

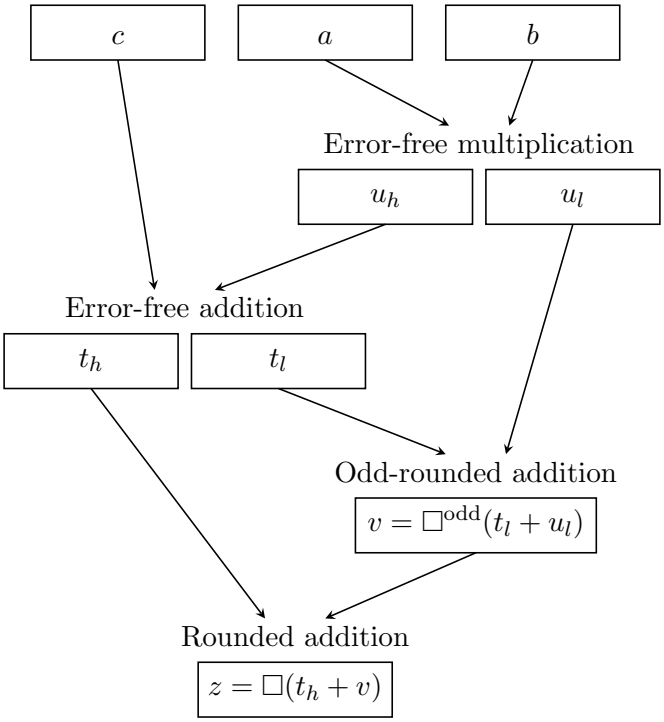


Figure 5.5 – Emulation of FMA: $z = \square(a \cdot b + c)$.

three floating-point numbers a, b, and c, the goal is to compute $\square(a \cdot b + c)$. Figure 5.5 presents the algorithm.

It starts by performing an error-free multiplication, which gives $u_h + u_l = a \cdot b$, assuming that $a \cdot b$ is not too small [Dek71]. Then it performs an error-free addition, which gives $t_h + t_l = u_h + c$ [Knu98]. At this point of the algorithm, we have $a \cdot b + c = t_h + t_l + u_l$. So, it is just a matter of summing the three numbers without losing any information. We do so by adding to t_h the sum v of t_l and u_l rounded to odd. To prove that this is correct, we exhibit an extended format such that the equality

$$\square(t_h + \square^{\text{odd}}(t_l + u_l)) = \square(\square^{\text{odd}}_{\text{ext}}(t_h + (t_l + u_l)))$$

holds. By applying Theorem 5.4, we thus have

$$\square(t_h + v) = \square(t_h + t_l + u_l) = \square(a \cdot b + c).$$

If an extended format with a precision at least twice is available (*e.g.*, *binary64* for a *binary32* FMA), then a much simpler algorithm can be used:

$$\square(a \cdot b + c) = \square(\square^{\text{odd}}_{\text{ext}}(\circ_{\text{ext}}(a \cdot b) + c)).$$

That is what Jelinek's implementation of `fmaf` does in the GNU C library and it explicitly cites our work as a reference.[3]

[3] `https://sourceware.org/git/?p=glibc.git;f=sysdeps/ieee754/dbl-64/s_fmaf.c;hb=HEAD`

5.3 Applications

To conclude this document, let us consider three examples that illustrate various uses of the presented concepts. The first application is the formal verification of a C function implementing a three-point numerical scheme for solving the wave equation. Section 5.3.1 focuses on the parts of that work that are related to the Coquelicot library and the Gappa tool.

The second application is an example of a mathematical conjecture that was disproved through massively parallel computations. Section 5.3.2 focuses on the part that relies on interval arithmetic and was implemented using Boost.Interval.

The third application is an ongoing work that has motivated several new features of Why3. The objective is to verify a WhyML library that implements some state-of-the-art algorithms for arbitrary-precision integer arithmetic and then to extract it to a C library, so as to get sensibly similar performance as GMP. Section 5.3.3 focuses on the way the memory is modeled and on the extraction to C.

5.3.1 Partial differential equations

The Coquelicot library has been presented in Section 3.1.2. The motivation for this formalization originated from the study of a numerical scheme for solving the unidimensional wave equation. This equation models the propagation of waves along an ideal vibrating elastic rope that is tied down at both ends. Its partial differential equation is obtained from Newton's laws of motion. For the sake of simplicity, the gravity is neglected, so the rope is supposed rectilinear when at rest. The rope is also supposed to have a constant propagation velocity c. Given a source term s, the so-called *wave equation* of the rope is thus

$$\frac{\partial^2 p}{\partial t^2} - c^2 \frac{\partial^2 p}{\partial x^2} = s.$$

The three-point numerical scheme corresponding to this partial differential equation is given by the C function of Figure 5.6, written by François Clément. This function returns a bidimensional array `p` of size $(i_{\max}+1) \times (k_{\max}+1)$, which discretizes the space and time with steps Δx and Δt, respectively. Let us denote $\tilde{p}_i^k$ the floating-point element `p[i][k]` of the array. Let us denote p_i^k the value that would have been obtained if `p[i][k]` had been computed with infinite precision. This value approximates the value $p(i \cdot \Delta x, k \cdot \Delta t)$ of the exact solution to the wave equation. The variables `ni`, `nk`, `dt`, `dx`, of the source code represent the values $i_{\max}$, $k_{\max}$, Δt, and $\Delta x = 1/i_{\max}$, respectively. The initial position of the rope is abstracted by the `p_zero` function, while the initial velocity and the source term s are set to zero, for simplicity. Both ends of the rope are tied, so $p_0^k = p_{i_{\max}}^k = 0$.

The three-point scheme is a very simple scheme that computes the new position p_i^{k+1} of the rope using its previous position at three different points. If we ignore the rounding errors due to the *binary64* computations, the scheme actually solves the following discrete equation for $k \in [1; k_{\max} - 1]$ and $i \in [1; i_{\max} - 1]$:

$$\frac{p_i^{k+1} - 2p_i^k + p_i^{k-1}}{\Delta t^2} - c^2 \frac{p_{i+1}^k - 2p_i^k + p_{i-1}^k}{\Delta x^2} = s_i^k.$$

The main question is: how well does the result of the `forward_prop` function approximates the exact solution p? There are two parts to this question. First, how large is the method error between $p(i \cdot \Delta x, k \cdot \Delta t)$ and the infinitely precise value p_i^k computed by the numerical

```
double **forward_prop
  (int ni, int nk, double dt, double v, double xs, double l)
{
  double **p;
  int i, k;
  double a1, a, dp, dx;

  dx = 1./ni;
  a1 = dt/dx*v;
  a = a1*a1;

  /* Allocate space-time variable for the discrete solution. */
  p = array2d_alloc(ni+1, nk+1);

  /* 1st initial condition and boundary conditions. */
  p[0][0] = 0.;
  for (i = 1; i < ni; i++)
    p[i][0] = p_zero(xs, l, i*dx);
  p[ni][0] = 0.;

  /* 2nd initial condition (p_one=0) and boundary conditions. */
  p[0][1] = 0.;
  for (i = 1; i < ni; i++) {
    dp = p[i+1][0] - 2.*p[i][0] + p[i-1][0];
    p[i][1] = p[i][0] + 0.5*a*dp;
  }
  p[ni][1] = 0.;

  /* Evolution problem and boundary conditions. */
  for (k = 1; k < nk; k++) {
    p[0][k+1] = 0.;
    for (i = 1; i < ni; i++) {
      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;
    }
    p[ni][k+1] = 0.;
  }
  return p;
}
```

Figure 5.6 – Numerical scheme for the unidimensional wave equation.

scheme? Second, how large is the round-off error between p_i^k and the *binary64* value $\tilde{p}_i^k$ actually computed by the C code? Under Sylvie Boldo's supervision, we worked on these two questions [BCF+10, BCF+13, BCF+14].

Real analysis

The convergence of the numerical scheme expresses the fact that the discrete solution gets closer to the continuous solution as the discretization steps Δx and Δt decrease to zero. It relies on the existence and regularity of the mathematical solution to the partial differential equation. In this specific case, this is quite simple as D'Alembert's formula is a solution of the unidimensional wave equation [D'A49]. Assuming that the source term s, the initial position p_0, and the initial velocity p_1, are regular enough, we have

$$p(x,t) = \frac{1}{2}\left(p_0(x+ct) + p_0(x-ct)\right) + \frac{1}{2c}\int_{x-ct}^{x+ct} p_1(\xi)\,d\xi + \frac{1}{2c}\int_0^t \int_{x-c(t-\tau)}^{x+c(t-\tau)} s(\xi,\tau)\,d\xi\,d\tau.$$

One way to verify that p is indeed a solution to the wave equation is to symbolically compute the differentials and to check that the partial differential equation holds. Unfortunately, this is next to impossible with the standard library of real analysis of Coq, hence the use of the Coquelicot library.

For that purpose, Catherine Lelay and I developed a Coq formalization of parametric integrals [BLM15]. Indeed, the fundamental theorem of calculus only handles the simplest cases, *e.g.*,

$$\left(x \mapsto \int_a^x f(t)dt\right)'(x) = f(x),$$

while we needed to prove that, under sufficient hypotheses, the following equality holds:

$$\left(x \mapsto \int_{a(x)}^{b(x)} f(x,t)dt\right)' = \int_{a(x)}^{b(x)} \frac{\partial f}{\partial x}(x,t)dt - f(x,a(x))\cdot a'(x) + f(x,b(x))\cdot b'(x).$$

When verifying D'Alembert's formula, applying this theorem manually is quite cumbersome, due to the nested integrals and the order-2 partial differential. So, Catherine Lelay and I also developed a reflection-based tactic for Coq that automatically performs symbolic differentiation, even in presence of parametric integrals [LM12]. This `auto_derive` tactic generates the side conditions that formally ensure that the differential is valid.

While we have exhibited a closed formula as the solution of the wave equation, the formula itself is not needed to prove the convergence of the numerical scheme. The only thing that matters is that the solution exists and is regular enough for us to apply Taylor-Lagrange's approximation theorem to it. Sylvie Boldo and I formally proved this theorem [BLM15]. Before stating its consequent, let us have a look at an important intermediate lemma: Schwarz' theorem. For the sake of readability, its Coq statement is not written *in extenso*. The • symbol denotes the position of the differentiation variable. For example, `(ex_derive f(•,v) u)` is short for `(ex_derive (fun z => f z v) u)`, *i.e.*, f is differentiable with respect to its first variable at point (u,v). Partial derivatives with respect to the first variable and to the second variables are written $\frac{\partial}{\partial 1}$ and $\frac{\partial}{\partial 2}$, respectively. Schwarz' theorem states that, if the order-2 mixed derivatives of a bivariate function f exist in a neighborhood of (x,y) and are continuous at (x,y), then these derivatives are equal at (x,y).

```
Lemma Schwarz : forall (f : R -> R -> R) (x y : R),
  locally_2d (fun u v =>
      ex_derive f(•,v) u /\ ex_derive f(u,•) v /\
      ex_derive ∂f/∂2(•,v) u /\ ex_derive ∂f/∂1(u,•) v
    ) x y ->
  continuity_2d_pt ∂²f/∂1∂2 x y -> continuity_2d_pt ∂²f/∂2∂1 x y ->
  ∂²f/∂1∂2(x,y) = ∂²f/∂2∂1(x,y).
```

Taylor-Lagrange's theorem then states that the following equality holds under some hypotheses on the regularity of the bivariate function f:

$$f(x', y') = \mathrm{Tpol}(f, n, (x, y), (x' - x, y' - y)) + \mathcal{O}(\|(x' - x, y' - y)\|^{n+1})$$

with

$$\mathrm{Tpol}(f, n, (x, y), (u, v)) = \sum_{p=0}^{n} \frac{1}{p!} \left(\sum_{m=0}^{p} \binom{p}{m} \cdot \frac{\partial^p f}{\partial x^m \partial y^{p-m}}(x, y) \cdot u^m v^{p-m} \right).$$

Note that it would have been impossible to state such a theorem, if not for the total functions provided by Coquelicot (see Section 3.1.2).

These few concepts of analysis are the basic blocks that we need, in order to prove the consistency, stability, and convergence of the numerical scheme [BCF+10]. These are intricate and tedious proofs which will not be explained here.

Round-off error

At this point, we only know that the numerical scheme is well-behaved, when evaluated using an infinitely precise arithmetic. We have no idea if the rounding errors due to floating-point arithmetic cause the computed solution to diverge from the exact solution. For all we know, the round-off error might grow as $O(2^k)$ with k the time index, which would make the numerical scheme useless in practice.

In addition to the assumptions on the initial velocity and the source term, we now suppose that the initial position is such that the position of the rope is never larger than 1. Note that this is a hypothesis on the exact solution of the wave equation. So, it is just a matter of scaling the original problem. But thanks to this hypothesis, we will be able to prove later that the computed solution of the numerical scheme is never larger than 2. We could prove a tighter bound, *e.g.*, 1.1, but it does not bring any improvement to the round-off error analysis.

Let us denote $\tilde{a}$ the value of variable `a`. Given the code of Figure 5.6, we know that the computed values $\tilde{p}_k^k$ satisfy the following equality, assuming the compiler did not mess too much with the code.

$$\tilde{p}_i^{k+1} = \circ \left[2\tilde{p}_i^k - \tilde{p}_i^{k-1} + \tilde{a} \cdot (\tilde{p}_{i+1}^k - 2\tilde{p}_i^k + \tilde{p}_{i-1}^k) \right].$$

We are interested in bounding the round-off error $\Delta_i^k = \tilde{p}_i^k - p_i^k$. We do so by proving the following invariant on the outer loop of the numerical scheme:

$$\forall i,\ |\Delta_i^k| \leq 78 \cdot 2^{-53} \cdot (k + 1) \cdot (k + 2). \tag{5.4}$$

The difficult part lies in verifying that this invariant is indeed preserved at each iteration. Let us denote δ_i^{k+1} the rounding error produced at index k. By definition, it satisfies the following equation.

$$\delta_i^{k+1} = \tilde{p}_i^{k+1} - (2\tilde{p}_i^k - \tilde{p}_i^{k-1} + a \cdot (\tilde{p}_{i+1}^k - 2\tilde{p}_i^k + \tilde{p}_{i-1}^k)).$$

To deduce a bound on δ_i^k, we have to estimate the range of $\tilde{p}_i^k$. From the assumption on the exact position of the rope, $|p(x,t)| \leq 1$, and from the stability of the scheme, we deduce $|p_i^k| \leq 1.5$, given some reasonable assumptions on the grid size. From the invariant, we then deduce that the values of $\tilde{p}$ used in the computation of $\tilde{p}_i^{k+1}$ are in the range $[-2;2]$, assuming $k_{\max}$ is not too large. From this enclosure of $\tilde{p}$, Gappa (see Section 4.2.1) easily produces a bound on δ_i^k by forward error analysis. For computations performed using *binary64* operations, we get $|\delta_i^k| \leq 78 \cdot 2^{-52}$.

The key insight is that, since the operations performed at each loop iteration are linear by nature, we have the following equalities.

$$\begin{aligned}
\Delta_i^{k+1} &= \tilde{p}_i^{k+1} - p_i^{k+1} \\
&= \left(2\tilde{p}_i^k - \tilde{p}_i^{k-1} + a \cdot (\tilde{p}_{i+1}^k - 2\tilde{p}_i^k + \tilde{p}_{i-1}^k)\right) - p_i^{k+1} + \delta_i^{k+1}, \\
&= \left(2\Delta_i^k - \Delta_i^{k-1} + a \cdot (\Delta_{i+1}^k - 2\Delta_i^k + \Delta_{i-1}^k)\right) + \delta_i^{k+1}.
\end{aligned}$$

In other words, Δ_i^k is a solution of the discrete scheme with a source term δ_i^k. Since the discrete scheme is linear, Δ_i^k is thus a linear combination

$$\Delta_i^k = \sum_{\ell \geq 0, j \in \mathbb{Z}} \lambda_j^\ell \cdot \delta_{i+j}^{k-\ell},$$

with λ_j^ℓ the fundamental solution of the discrete scheme. From then on, a study of this fundamental solution shows that $\sum_j |\lambda_j^\ell| = \ell+1$, from which we deduce that the invariant (5.4) is preserved [BCF+14].

In the end, to tie all these proofs together, the C code of Figure 5.6 was fully annotated with a specification stating how the method and round-off errors are bounded. Verification conditions were then generated using Frama-C/Jessie/Why3. They were discharged using either Gappa or Coq.

5.3.2 Computational mathematics

Let us look at a second application. As mentioned in Section 2.2.3, incompleteness of interval arithmetic makes it generally ill-suited for verifying equalities. *A contrario*, it is well suited for disproving equalities. Let us illustrate it with Gramain's conjecture about the value of Masser-Gramain's constant. First, we remind the definition of Euler-Mascheroni's constant:

$$\gamma = \lim_{n \to \infty} \left(\sum_{k=2}^{n} \frac{1}{k-1} - \log n \right).$$

The value $k-1$ can be seen as the length of the smallest segment containing k points of the real line with integer coordinates, named *integer points* for short. The γ constant can

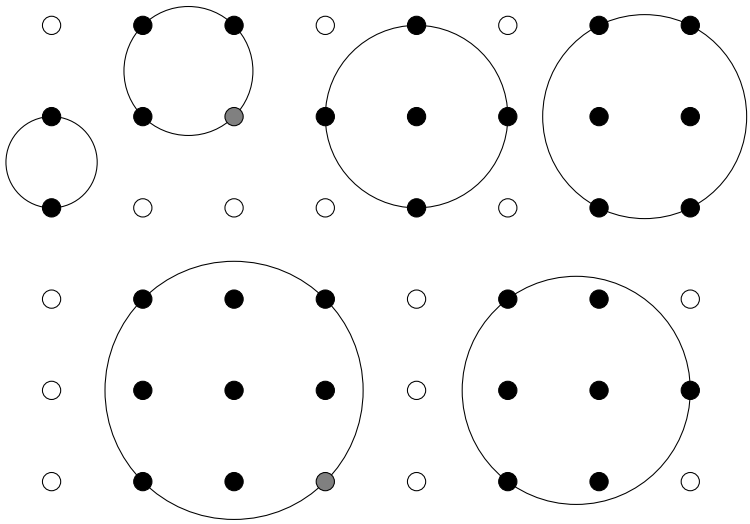


Figure 5.7 – Smallest disks containing up to $k = 9$ integer points.

be generalized to higher dimensions. Replacing segments by disks in the bidimensional plane gives Masser-Gramain's constant:

$$\delta = \lim_{n\to\infty} \left(\sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log n \right),$$

where r_k is the radius of the smallest disk containing k integer points of the plane, *i.e.*, Gaussian integers. Constant δ is used to qualify the exponential growth of non-polynomial entire functions that are stable on the set of Gaussian integers. Figure 5.7 shows some minimal disks for the first values of k. The corresponding values of r_k^2 are

$$r_2^2 = \frac{1}{4}, \quad r_3^2 = r_4^2 = \frac{1}{2}, \quad r_5^2 = 1, \quad r_6^2 = \frac{5}{4}, \quad r_7^2 = \frac{25}{16}, \quad r_8^2 = r_9^2 = 2.$$

Notice that all the squared radii are rational numbers. This should not come as a surprise, since the smallest disk containing integer points necessarily have three integer points on its circumference (or two diametrically opposed points in some degenerate cases). This means that one can compute the exact value of the sum $\sum_{k\leq n} r_k^{-2}$. In fact, thanks to some intensive computations to compute the first radii r_k and an analytic bound on the remainder of the series, Gramain and Weber obtained the first two decimals of δ: $1.811447299 < \delta < 1.897327117$, which was compatible with a closed formula conjectured by Gramain [GW85].

The issue is that the time complexity of computing r_k grows faster than $k^{5/3}$, since one needs to enumerate all pair of integer points (the third point can be fixed at origin), and for each pair, enumerate all the integer columns to count the integer points on it, with an ever increasing cost per integer column, due to accuracy issues. For example, computing r_{10^6} takes several minutes. It is thus impossible to compute the sum for large values of n, yet this is needed due to the poor accuracy of the analytic bounds on the remainder of the series.

To solve this issue, we devised a mixed approach. Through massively parallel computations, Paul Zimmermann was able to compute the values of r_k for $k < 10^6$. Using Poisson's formula and some Fourier analysis, George Nowak found some improved analytic bounds

$S \leftarrow \{([0;0.5],[0;0.5],\mathbf{z})\}$ with $\mathbf{z}$ a poor enclosure of r_k
$\mathbf{r} \leftarrow +\infty$
while $S \neq \emptyset$, **do**
  extract an element $(\mathbf{x},\mathbf{y},\mathbf{z})$ from S
  if $\overline{y} < \underline{x}$ or $\underline{r} \leq \underline{z}$ or nb_points$(\mathbf{x},\mathbf{y},\underline{r}) < k$, **then continue**
  increase $\underline{z}$ while keeping nb_points$(\mathbf{x},\mathbf{y},\underline{z}) < k$
  decrease $\overline{z}$ while keeping nb_points$(\{\underline{x}\},\{\underline{y}\},\overline{z}) \geq k$
  if $\mathbf{x} \times \mathbf{y}$ is small enough, **then**
    $\mathbf{r} \leftarrow \min(\mathbf{r},\mathbf{z})$
  else
    split $\mathbf{x} \times \mathbf{y}$ into two smaller rectangles $\mathbf{x}' \times \mathbf{y}'$ and $\mathbf{x}'' \times \mathbf{y}''$
    insert $(\mathbf{x}',\mathbf{y}',\mathbf{z})$ and $(\mathbf{x}'',\mathbf{y}'',\mathbf{z})$ into S
  endif
done
return $\mathbf{r}$

Figure 5.8 – Computing an enclosure of r_k given k.

that give r_k^2 with a relative accuracy of $O(k^{-2/3})$, which made it usable for $k \geq 10^9$. Finally, I ran some massively parallel interval computations to obtain an approximation of r_k for $10^6 \leq k < 10^9$ [MNZ13].

This approximation program was written in C++ using the Boost.Interval library. Its algorithm is sketched in Figure 5.8. Given an integer k and a set C, the objective is to find an enclosure of the radius of the smallest disk containing k integer points whose center is contained in C. Note that, by symmetry and translation, applying this algorithm to the triangle $C = \text{hull}\{(0,0),(0.5,0.5),(0,0.5)\}$ suffices to find an enclosure of r_k. The algorithm starts from an overapproximation $(\mathbf{x},\mathbf{y})$ of C and $\mathbf{z}$ of r_k. It then proceeds by maintaining a set $S = \{(\mathbf{x}_i,\mathbf{y}_i,\mathbf{z}_i)\}$ of pieces that still need to be considered, with the following invariant: the radius of the smallest disk containing k integer points with a center enclosed in $\mathbf{x}_i \times \mathbf{y}_i$ is enclosed in $\mathbf{z}_i$, assuming there is one. Pieces of the set S are either discarded or replaced by smaller pieces, until the pieces are small enough to provide a tight enclosure of r_k.

Let us denote $\mathbf{r}$ the enclosure of the currently known minimum, *i.e.*, either it tightly encloses r_k, or r_k is attained over one of the remaining pieces of S. A piece $(\mathbf{x},\mathbf{y},\mathbf{z})$ of S gets discarded early if we know that the minimal radius over $\mathbf{x} \times \mathbf{y}$ is larger than $\underline{r}$. This happens if $\underline{r} \leq \underline{z}$ holds. This also happens if all the disks of center in $\mathbf{x} \times \mathbf{y}$ and of radius $\underline{r}$ contain at most $k-1$ integer points. Thus, we need some way to enclose the number of integer points for a disk of radius r:

$$\text{nb\_points}(\mathbf{x},\mathbf{y},r) = \sum_{j=\lceil \underline{x}-r \rceil}^{\lfloor \overline{x}+r \rfloor} \lfloor \mathbf{y}+\mathbf{h}_j \rfloor - \lceil \mathbf{y}-\mathbf{h}_j \rceil + 1 \qquad \text{with } \mathbf{h}_j = \sqrt{r^2-(\mathbf{x}-j)^2}.$$

If a piece cannot be discarded, we need to replace it with some smaller pieces. Splitting a piece with respect to $\mathbf{x}$ and $\mathbf{y}$ is easy, as a plain bisection suffices. For the $\mathbf{z}$ part, we need a more subtle approach, in order to preserve the above invariant. So, rather than splitting $\mathbf{z}$, we refine it. Using a bisection, we increase the lower bound $\underline{z}$, as long as all the disks of center

in $\mathbf{x} \times \mathbf{y}$ contain at most $k-1$ integer points. To refine the upper bound $\overline{z}$, we just consider one arbitrary center in $\mathbf{x} \times \mathbf{y}$ and again proceed by bisection. Note that the naive analytic bound $r_k \leq \sqrt{(k-1)/\pi}$ is surprisingly accurate, so the algorithm does not try hard to refine this upper bound.

In the end, we found the following enclosure: $1.81977613409613 < \delta < 1.81983226978634$, which disproves Gramain's conjecture. Note that none of that work was formally verified. In particular, the analytic bounds mix various topics of geometry and analysis, which would be interesting to formalize. They are still out of scope of existing formal developments, though.

5.3.3 Extracting WhyML programs to C

Finally, let us look at a third application. The GNU Multi-Precision library has been summarily presented in Section 5.1.2. It offers an arithmetic operators on arbitrarily large integers. For maximal performance, it uses numerous state-of-the-art algorithms for basic operations like addition, multiplication, and division; these algorithms are selected depending on the sizes of the numbers involved. Moreover, the implementation is written in low-level C code, and depending on the target computer architecture, some parts are even rewritten in Assembly.

Being highly optimized for run-time efficiency, the code of GMP is intricate and thus error-prone. It is extensively tested but it is hard to reach a satisfactory coverage in practice: the number of possible inputs is very large, the different branches of the algorithms are numerous, and some of them are taken with a very low probability (some branches are taken with probability 2^{-64} or less). Bugs in the division, occurring with very low probability, were discovered in the past.[4] Verifying the code for all inputs using static program verification is thus desirable. Such a verification, however, is difficult, not only because of the intrinsic complexity of the algorithms, but also because the code is written in a low-level language with performance in mind, but not verification.

To alleviate this issue, Raphaël Rieu-Helft translated GMP's algorithms into WhyML and verified their correctness using Why3.[5] With the collaboration of Claude Marché, we then devised a mechanism to turn WhyML code into C code, so as to obtain a fully verified library, compatible with GMP (*i.e.*, function signatures are the same), and almost as efficient as GMP on medium-sized integers (*i.e.*, up to around 20 words of 64 bits), as long as GMP's algorithms are not directly written in Assembly [RHMM17].

There are two main challenges. The first one, which was alluded in Section 5.1.2, is to actually succeed in verifying the intricate algorithms. The second one is to convert some high-level WhyML code into an efficient executable code. Our approach was to first design a dedicated *memory model* in Why3, on top of which we then implemented our functions. This memory model is designed to permit a direct compilation from WhyML to C. In particular, GMP's algorithms critically rely on features of the C language such as pointers, so we need to encode them.

Figure 5.9 shows GMP's code for the comparison of two integers that have the same size, once it has been cleaned from all the macros and beautified a bit. Integers from the `mpn` component are stored as consecutive unsigned words in memory, called *limbs*, in a little-endian fashion. The input integers are represented by pointers to their least-significant word (`xp` and `yp`), as well as the number of limbs they are comprised of (`size` for both of them). Note that there is no requirement that either pointer points at the start of an allocated memory block;

[4] Look for 'division' at `https://gmplib.org/gmp5.0.html`.

[5] `http://toccata.lri.fr/gallery/multiprecision.en.html`

```
int mpn_cmp(mp_srcptr xp, mp_srcptr yp, mp_size_t size)
{
  /* ASSERT (size >= 0); */
  mp_size_t i = size;
  while (--i >= 0) {
    mp_limb_t x = xp[i];
    mp_limb_t y = yp[i];
    if (x != y) {
      /* Cannot use x - y, may overflow an ''int'' */
      return x > y ? 1 : -1;
    }
  }
  return 0;
}
```

Figure 5.9 – GMP's C code for comparing integers.

they can point at any position of a block, as long as the blocks are large enough to contain `size` words from this position. For example, if the caller already knows that the first words are equal, it can pass some shifted pointers in order to avoid useless computations.

To translate this code to WhyML as faithfully as possible, we need some type to represent pointers. Notice that this code (as well as GMP in general) does not make use of pointer casts. It does not need the C address-of operator `&` either. Generally speaking, we do not use a memory model that would cover all features of C, because we want to benefit from the non-aliasing properties provided by Why3's type system. The benefit is that both the specifications and the proofs are simpler. With a general model of C heap memory, we would need to state a lot of non-aliasing hypotheses among the pointers. These properties would generate extra verification conditions to be established by backend provers. Moreover, the other verification conditions would be more difficult to discharge.

The C heap memory is modeled as a set of memory blocks called *objects* in the C99 standard [ISO11, §3.14]. The WhyML polymorphic type `ptr 'a` represents pointers to blocks storing consecutive data of type `'a`. The `data` field of a pointer is an array containing the block content, while the `offset` field indicates which array cell it points to.

```
type ptr 'a = abstract { mutable data: array 'a; offset: int }
```

Let us illustrate the memory model with two primitives. The `get_ofs` function takes a pointer `p` and a signed integer `ofs` as arguments and it returns the value stored in the corresponding cell, *i.e.*, `*(p + ofs)` in C.

```
val get_ofs (p: ptr 'a) (ofs: int32): 'a
  requires { 0 <= p.offset + ofs < p.data.length p }
  ensures  { result = p.data[p.offset + ofs] }
```

The precondition following `requires` states that `(p + ofs)` should point inside the memory block. The postcondition following `ensures` states that the returned value is stored in the `data` array associated to `p`.

The `incr` function takes a pointer `p` and a signed integer `ofs` as arguments and it returns a pointer that points to `(p + ofs)`.

```
val incr (p: ptr 'a) (ofs: int32): ptr 'a
  requires { 0 <= p.offset + ofs <= p.data.length p }
  alias    { p.data with result.data }
  ensures  { result.offset = p.offset + ofs }
  ensures  { result.data = p.data }
```

The precondition states that, as in C, the operation is well defined only if the new pointer still points in the same memory block (or one past the end). The returned pointer has its own `data` array, so the second postcondition states that it contains the same values as the array of the original pointer. That is not sufficient, though. Indeed, by default, Why3 supposes that the input pointer and the output pointer have non-aliasing `data` fields. So, writing through one of the pointers would not affect reading through the other one. Thus, we have implemented an `alias` directive for Why3, which we use to indicate that both pointers share the same array.

The memory model provides several other operations, including write operations, which are all mapped trivially to C operations. As for the `valid` predicate, it characterizes that a pointer `p` is suitable to access the next `sz` cells from the array it points to.

```
predicate valid (p: ptr 'a) (sz: int) =
  0 <= sz /\ 0 <= p.offset /\ p.offset + sz <= p.data.length p
```

Figure 5.10 shows the WhyML code for comparing two `mpn` integers of the same size, as well as its specifications and some annotations needed to prove its correctness. The code is a bit verbose but it is mostly a one-to-one translation of the C code of Figure 5.9.

The preconditions of the function indicate that the pointers shall be valid at the time the function is called. The postcondition indicates that the value returned by the function is equal to the result of `compare_int` applied to the integers represented by both pointers. The variant of the loop guarantees that it terminates, since index i decreases at each step, according to the natural well-founded ordering on type `int32`. The invariant states that the limbs of both integers at index i or larger are equal. All the assertions needed to guide the SMT solvers toward the proof of the verification conditions have been removed, for the sake of readability.

Figure 5.11 shows the C code obtained after extraction, with superfluous parentheses removed for readability. An optimizing compiler would compile this code to the same Assembly code as it would for the original function of Figure 5.9.

The extraction to C supports only part of WhyML's features. For example, recursive datatypes are not supported, as they would require some automatic memory management. Exceptions are not supported either, except for some specific constructs. If an exception is only caught at the end of a function and if the value it carries acts as the return value, then any `raise` in the body of the function is extracted as `return`. Why3 recognizes a similar idiom for loops, so as to produce `break` in that case [RHMM17].

Comparison is the simplest of the functions implemented in WhyML. Our verified library also implements addition, subtraction, and shifts. Regarding multiplication and division, we have implemented and verified GMP's *schoolbook* algorithms. For multiplication, this is the naive quadratic algorithm. For division, the algorithm is also quadratic, but much more intricate than anything one could ever find in an actual schoolbook [MG11]. Finally, Raphael

```
function compare_int (x y: int): int =
  if x < y then -1 else if x = y then 0 else 1

let wmpn_cmp (x y: ptr limb) (sz: int32): int32
  requires { valid x sz }
  requires { valid y sz }
  ensures  { result = compare_int (value x sz) (value y sz) }
= let ref i = sz in
  while i >= 1 do
    variant { i }
    invariant { 0 <= i <= sz }
    invariant { forall j. i <= j < sz ->
                x.data[x.offset+j] = y.data[y.offset+j] }
    i := i - 1;
    let lx = get_ofs x i in
    let ly = get_ofs y i in
    if lx <> ly then
      return (if lx > ly then 1 else -1)
  done;
  0
```

Figure 5.10 – WhyML code for comparing integers.

```
int32_t wmpn_cmp(uint64_t *x, uint64_t *y, int32_t sz) {
  int32_t i;
  int32_t o;
  uint64_t lx, ly;
  i = sz;
  while (i >= 1) {
    o = i - 1;
    i = o;
    lx = *(x + i);
    ly = *(y + i);
    if (!(lx == ly)) {
      if (lx > ly) return 1;
      else return -1;
    }
  }
  return 0;
}
```

Figure 5.11 – C code for comparing integers, once extracted from WhyML.

Rieu-Helft also verified a subquadratic multiplication based on two Toom-Cook algorithms: Toom-2 and Toom-2.5.

These two divide-and-conquer algorithms have an interesting property. Pointers to a large temporary buffer are passed to them. This buffer, as well as the output block, are dynamically split into smaller parts, where the recursive calls perform their temporary computations and store their results. It is thus important to express that the result of a recursive call is not overwritten by a subsequent recursive call until it has been fully used. This makes the situation akin to having a single heap and to manually handling non-aliasing of pointers, which is what we wanted to avoid in the first place. So, we have enriched our memory model to support this use case too. This makes the memory model much more difficult to grasp. In particular, it is no longer possible, at first glance, to be convinced that it is sound. So, future works will be dedicated to developing methods for verifying the soundness of *ad hoc* memory models, if possible in Why3.

Regarding performance, our verified library is almost competitive with GMP and it could be used as a drop-in replacement in some cases. The "almost" part resides in the way the two libraries are compared. First, our library is nowhere as fast as the hand-written Assembly code of GMP, as it does not have access to some dedicated processor opcodes, *e.g.*, manipulation of carry bits. It is only against the pure C functions of GMP that we can compete. Second, we have implemented and verified only a small part of GMP's algorithms. In particular, when the numbers are sufficiently large, GMP switches to other algorithms, *e.g.*, a divide-and-conquer division, which means that our library is no longer competitive. But large numbers are not the only case. For example, GMP is also using a dedicated algorithm when both division operands have almost the same small size, while we are using the generic algorithm, which is a bit slower. Still, for small to medium-sized integers (*e.g.*, less than 2000 bits), we offer a formally verified library that is only 5% to 10% slower than GMP on average.

Chapter 6

Perspectives

Sections 3.1.3, 3.2.4, 3.3.4, 3.4.3, 3.5.2, 4.2.4, 4.3.3, 4.4.3, and 4.5.3, have already explained, for each part of my research, how it compares to related works, what are its shortcomings, and how it could be improved further. I will not rehash all these details here. Instead, this short chapter will give an overview of the directions in which my research work might progress in the next few years.

6.1 Proof assistants and mathematics

Despite its tediousness and the expertise it requires, formal verification has gained a foothold in a few domains. For example, the use of formal proofs is well established when it comes to research related to programming languages. In fact, recent papers that do not come with a formal development might look suspicious. This led Robert Constable to half-jokingly say, during his Herbrand Award's talk, that any submission to the *Principle of Programming Languages* conference should be accompanied with a Coq proof.

The situation in other domains of computer science does not look that good, though, as pen-and-paper proofs are still the standard way of guaranteeing results. Formal proofs are seen as pointless efforts, if not downright disparaging. Indeed, if the authors succeeded in formalizing their result, it must mean that their result was not that interesting, doesn't it? And despite some breakthroughs like the odd-order theorem or Kepler's conjecture, the same mindset can be found among mathematicians. Yet, computer scientists and mathematicians are not reluctant to use tools to increase the confidence in their results. For example, they might launch some computer algebra system to perform part of their proofs: optimizing a function, computing an integral, finding eigenvalues, simplifying an expression, and so on. They might even implement their own routine if the computer algebra system does not suffice. And if they are proponents of reproducible research, they will carefully document the whole process that led them to their results.

Today, users of proof assistants and users of computer algebra systems are two mostly separate communities. Bringing them closer is no small undertaking. I am mostly interested in getting users of computer algebras to use proof assistants instead. As far as they are concerned, this usually means forsaking a rich, fast, and user-friendly tool, and the extra confidence in the results might not seem worth the effort. Thus, it is important to make proof assistants much more approachable. Note that the goal is not to force users to formally prove

theorems (or at least not yet), but just to get them accustomed to running formally verified algorithms whenever possible.

Regarding the ease of use, Coquelicot was a first step in this direction. Indeed, the motivation for this library was to make it possible for Coq users to write limits, power series, derivatives, integrals, and so on, almost the same way as they would on paper, *i.e.*, without having to struggle with dependent types and embedded proofs (see Section 3.1). The library then moved to make the proofs less specific to real numbers, so that the theorems could also be used in other contexts, *e.g.*, complex analysis, without having to start from scratch again. The various works on reification, *e.g.*, in CoqInterval, are also a way to improve ease of use, as the user no longer needs to accommodate the input format of the verified algorithms. So, things are rather well understood with respect to the design of an expressive input language. We are now at a point where this is mostly a matter of finding the proper formalism to express additional theories, *e.g.*, linear algebra, and then doing a lot of tedious work to formally verify the corresponding lemmas. In particular, Coq is critically lacking a library that would provide all the elementary and special functions one has come to expect from a computer algebra system, *e.g.*, Bessel functions. This needs to be addressed.

Regarding output, the situation, however, is close to disastrous. Indeed, formal systems are designed to verify proofs. As such, the feedback they give to a user hardly exceeds a single bit of information, *i.e.*, "this proof is (in)correct". For example, consider the polynomial equality $(x+1)(x-1) = x^2 - 1$. Proving that it holds using Coq is straightforward; it is just a matter of stating this equality and then typing `ring`. Now, assume that the user does not know yet about $x^2 - 1$. So, the user wants the tool to compute what $(x+1)(x-1)$ expands to. It happens that the verified algorithm at the heart of the `ring` tactic can do so. But, the user now has to write a dummy theorem statement, *e.g.*, $(x+1)(x-1) = y \Rightarrow \mathit{True}$, just so that the algorithm can be instantiated. This is an ergonomic failure. But it gets worse. Assume that the user wants to plot $(x+1)(x-1)$ for $x \in [-10; 10]$, because it might give some insight on a proof. One could easily turn some algorithms of CoqInterval into a reliable plotter *à la* Sollya. But then, how to use them? State a dummy theorem, start a proof, call the algorithms, obtain a list of segments, copy-paste them into a file, put the file into the proper format, and finally call an actual plotter on it. Expecting users to go through this obstacle course for such a basic feature is not reasonable. So, there is a lot of research to be done on how users can interact with proof assistants.

Once the users have a rich formalism to express their formulas and a proper interface to query the system, the next step is to provide them with useful tools. On the symbolic side, Coq's standard library comes with a few decision procedures. On the numerical side, CoqInterval provides algorithms for computing interval and polynomial enclosures, and for performing proper and improper definite integration (see Section 3.5). There are many other procedures that should be implemented to provide a minimal working environment. In particular, there is currently no way for the user to define numbers or functions in an implicit way, *e.g.*, by an ordinary differential equation, and then to manipulate them as if they were closed expressions. That is a situation I want to remedy, *e.g.*, by leveraging the polynomial approximations computed by CoqInterval (see Section 3.4).

Finally, the last requirement is speed. Users might not be willing to wait hundreds of seconds for a numerical result that a computer algebra system would have returned in a split second, even if they have absolutely no guarantee on the latter. There are two main issues. First, performing computations inside a proof assistant might be several orders of magnitude slower than in a dedicated tool. There are some seemingly obvious solutions to

this first issue but they still have to be investigated, *e.g.*, letting the system access hardware resources such as floating-point units, or extracting and running verified algorithms. The second cause of slowdown is more insidious. When designing a formally verified procedure, one not only pays the cost of implementing the algorithm in the logic of the system, but one also has to pay the cost of verifying the algorithm. As a consequence, one might end up implementing a simpler but much less efficient algorithm. A potential solution is to turn to certificate-producing algorithms. They do not have to be verified and can run outside the formal system at full speed. Only the certificate checker has to be verified and run inside the system. But that approach is worth only if this checker is simpler and faster than the solving algorithm. Unfortunately, few problems are known to be solvable that way and further research is required. Of particular interest are methods based on contracting iterations, as it might be possible to perform only the last iteration in the formal system, in a naive way.

6.2 Deductive program verification for the masses

The Coq system is well suited for proving complicated mathematical theorems with a high level of trust. For simpler properties, however, its expressiveness and lack of automation are more of a hindrance for a non-expert user. I am particularly interested in the proof obligations that show up when trying to verify the correctness of an algorithm. Consider for example an implementation of *quicksort* to sort an array of integers. This is a simple and useful algorithm, yet it is easy to get it wrong because of an off-by-one error. In other words, there is no real challenge with respect to verification, but we still want the computer to assist us in making sure that the devil is not hidden in the details of the implementation. If we were to perform this verification using Coq, we would surely succeed, but the amount of work would be disproportionate with respect to the difficulty of the proof. Indeed, we would first have to embed this imperative and effectful algorithm into the logic of Coq. So, before we had even started the actual verification, we would already be hitting a wall.

That is why I am co-developing the Why3 tool (see Section 2.3.1). By focusing on a custom programming language that supports contracts, the matter of embedding the code into the logic of a formal system vanishes partially. The downside is that the user has to learn yet another language (either that or use Why3 as the backend for an existing language). Thus, we are putting a lot of care into making the WhyML language as expressive as possible. In particular, because the core motivation is deductive program verification, there are some inherent limitations when it comes to implementing programs with aliases. The way we have handled pointers and pointer arithmetic during our verification of GMP's algorithms was a first step toward hiding these limitations from the user (see Section 5.3.3). I intend to keep researching on expressive memory models that do not get in the way of the user, neither during the implementation of an algorithm nor during its verification.

Once an algorithm has been implemented and annotated with its specification, the next step is the verification. In an ideal world, the automated solvers called by Why3 would discharge all the proof obligations and the algorithm would be verified, or they would exhibit counterexamples that show the inadequacy between the code and the specification. In practice, the user is left with several proof obligations. In that case, the user can tell Why3 to send these remaining obligations to interactive systems, *e.g.*, Coq, but for a user who did not want to use Coq in the first place, that is a non-solution. Our work on reflection-based proofs is a way to solve this issue without leaving the confines of Why3 (see Sections 2.3.2

and 5.1.2). I intend to further explore this approach, to make it applicable to a larger range of proof obligations.

Another way to increase the amount of discharged proof obligations is to improve the interface with the solvers and to improve the solvers themselves. An example is floating-point arithmetic. A dedicated theory was added to Why3 and it maps to the corresponding support from SMT solvers when it exists. This was not the case of Alt-Ergo, so a floating-point theory was added (see Section 4.5). Unfortunately, despite several attempts, intricate proof obligations mixing both floating-point and real numbers cannot yet be automatically discharged by SMT solvers. These obligations fall into the logical fragment supported by Gappa (see Section 4.2), but only if the user has separated from them everything unrelated, *e.g.*, array accesses, and has instantiated the needed lemmas from the proof context. So, I still intend to research a theory of floating-point and real numbers that works in the context of SMT solvers while being as effective as Gappa's theory. That being said, Gappa is not the definitive target. Indeed, even if Gappa is still unrivaled on the most intricate proof obligations, especially when one wants to obtain a formal proof at the end, the use of symbolic affine forms for representing round-off errors has shown that it was possible to significantly improve the accuracy on error bounds on more straightforward goals. So, I also want to explore this kind of methods, as they might considerably reduce the amount of preliminary work from the user when trying to discharge a proof obligation related to round-off errors.

The long-term objective is to make it possible for a user to write an intricate floating-point algorithm in WhyML, annotate it with a precise specification, automatically discharge the vast majority of the proof obligations with no extra annotations, and finally have only the most fundamental behavioral properties left to prove. If we can get the users to that point, then they might be willing to go the extra mile, *i.e.*, add a few cut annotations or turn to Coq and Flocq, in order to fully complete the verification of their algorithm. This algorithm could then be extracted to C (or any other language supported by Why3) and made part of a library of verified floating-point functions.

Bibliography

[ABC18] ABC tech & strats. 2018. URL: https://docs.google.com/document/d/1cJfwdju9-6yTGemBhtZrjoDhWWDYj69OpzIg8D6w9R4.

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In M. Kaufmann and L. C. Paulson, editors, *1st Interactive Theorem Proving Conference (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98, Edinburgh, UK, July 2010. doi:10.1007/978-3-642-14052-5_8, hal:inria-00502496.

[Bar89] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989. doi:10.1109/32.24710.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. 472 pages. Texts in Theoretical Computer Science. Springer-Verlag, 2004. doi:10.1007/978-3-662-07964-5.

[BCC+12] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In B. Gramlich, D. Miller, and U. Sattler, editors, *6th International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Artificial Intelligence*, pages 67–81, Manchester, UK, June 2012. doi:10.1007/978-3-642-31365-3_8, hal:hal-00687640.

[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 M. Kaufmann and L. C. Paulson, editors, *1st Interactive Theorem Proving Conference (ITP)*, volume 6172 of *Lecture Notes on Computer Science*, pages 147–162, Edinburgh, UK, July 2010. doi:10.1007/978-3-642-14052-5_12, hal:inria-00450789.

[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, 2013. doi:10.1007/s10817-012-9255-4, hal:hal-00649240.

[BCF+14] 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 & Mathematics with Applications*, 68(3):325–352, 2014. doi:10.1016/j.camwa.2014.06.004, hal:hal-00769201.

[BDG+14] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014. doi:10.1007/s10703-013-0203-7.

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

[Ber12] Sergeï N. Bernstein. Sur l'ordre de la meilleure approximation des fonctions continues par les polynômes de degré donné. *Mémoire de la Classe des Sciences de l'Académie Royale de Belgique*, 4:1–103, 1912.

[Bes07] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In T. Altenkirch and C. McBride, editors, *International Workshop on Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62, Nottingham, UK, April 2007. doi:10.1007/978-3-540-74464-1_4.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *16th Calculemus Symposium*, volume 5625 of *Lecture Notes in Artificial Intelligence*, pages 59–74, Grand Bend, ON, Canada, July 2009. doi:10.1007/978-3-642-02614-0_10, hal:inria-00432726.

[BFM+13] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. In E. Cohen and A. Rybalchenko, editors, *5th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*, pages 191–201, Menlo Park, CA, USA, May 2013. doi:10.1007/978-3-642-54108-7_10, hal:hal-00875395.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *1st International Workshop on Intermediate Verification Languages (Boogie)*, pages 53–64, Wroclaw, Poland, August 2011. hal:hal-00790310.

[BJ10] Nicolas Brisebarre and Mioara Joldeş. Chebyshev interpolation polynomial-based tools for rigorous computing. In *35th International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 147–154, Münich, Germany, July 2010. doi:10.1145/1837934.1837966, hal:ensl-00472509.

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

A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, editors, *21st IEEE Symposium on Computer Arithmetic (Arith)*, pages 107–115, Austin, TX, USA, June 2013. doi:10.1109/ARITH.2013.30, hal:hal-00743090.

[BJLM15] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015. doi:10.1007/s10817-014-9317-x, hal: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 A. Goodloe and S. Person, editors, *4th International Symposium on NASA Formal Methods (NFM)*, volume 7226 of *Lecture Notes in Computer Science*, pages 85–99, Norfolk, VA, USA, April 2012. doi:10.1007/978-3-642-28891-3_9, hal:ensl-00653460.

[BKSF59] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic differentiation for the machine BESM. Technical report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959.

[BLM12] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Improving real analysis in Coq: a user-friendly approach to integrals and derivatives. In C. Hawblitzel and D. Miller, editors, *2nd International Conference on Certified Programs and Proofs (CPP)*, volume 7679 of *Lecture Notes in Computer Science*, pages 289–304, Kyoto, Japan, December 2012. doi:10.1007/978-3-642-35308-6_22, hal:hal-00712938.

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. doi:10.1007/s11786-014-0181-1, hal:hal-00860648.

[BLM16] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, 26(7):1196–1233, 2016. doi:10.1017/S0960129514000437, hal:hal-00806920.

[BM05] Sylvie Boldo and Guillaume Melquiond. When double rounding is odd. In *17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005. hal:inria-00070603.

[BM08] Sylvie Boldo and Guillaume Melquiond. Emulation of a FMA and correctly-rounded sums: Proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, 2008. doi:10.1109/TC.2007.70819, hal:inria-00080427.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In E. Antelo, D. Hough, and P. Ienne, editors, *20th IEEE Symposium on Computer Arithmetic (Arith)*, pages 243–252, Tübingen, Germany, July 2011. doi:10.1109/ARITH.2011.40, hal:inria-00534854.

[BM13] Sylvie Boldo and Guillaume Melquiond. Arithmétique des ordinateurs et preuves formelles. In P. Langlois, editor, *Informatique Mathématique : une photographie en 2013*, pages 189–220. Presses Universitaires de Perpignan, 2013. hal:hal-01767900.

[BM17] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs. Verifying Floating-point Algorithms with the Coq System.* 326 pages. ISTE Press – Elsevier, 2017.

[BMP03] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The Boost interval arithmetic library. In *5th Conference on Real Numbers and Computers (RNC)*, pages 65–80, Lyon, France, 2003. hal:inria-00348711.

[BMP06a] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. Bool_set: multi-valued logic. Technical Report 2136, ISO C++ Standardization Committee, 2006. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2136.pdf.

[BMP06b] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the Boost interval arithmetic library. *Theoretical Computer Science*, 351:111–118, 2006. doi:10.1016/j.tcs.2005.09.062, hal:inria-00344412.

[BMP06c] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. A proposal to add interval arithmetic to the C++ standard library. Technical Report 2137, ISO C++ Standardization Committee, 2006. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2137.pdf.

[BMP06d] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. Proposing interval arithmetic for the C++ standard. In *12th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN)*, Duisburg, Germany, 2006.

[BMSU86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 1–15, Cambridge, MA, USA, 1986. doi:10.1145/6012.15399.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *3rd International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529, Sendai, Japan, September 1997. doi:10.1007/BFb0014565.

[BRT18] Yves Bertot, Laurence Rideau, and Laurent Théry. Distant decimals of π: Formal proofs of some algorithms computing them and guarantees of exact computation. *Journal of Automated Reasoning*, 61(1):33–71, 2018. doi:10.1007/s10817-017-9444-2, hal:hal-01582524.

[BTRW15] Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In J.-M. Muller, A. Tisserand, and J. Villalba, editors, *22nd IEEE Symposium*

on Computer Arithmetic, pages 160–167, Lyon, France, June 2015. doi:10.1109/ARITH.2015.26.

[BW16] Martin Bromberger and Christoph Weidenbach. Fast cube tests for LIA constraint solving. In N. Olivetti and A. Tiwari, editors, *8th International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Artificial Intelligence*, pages 116–132, Coimbra, Portugal, June 2016. doi:10.1007/978-3-319-40229-1_9, hal:hal-01403200.

[Car37a] Henri Cartan. Filtres et ultrafiltres. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, 205:777–779, July 1937.

[Car37b] Henri Cartan. Théorie des filtres. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, 205:595–597, July 1937.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. Sethi, editor, *4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, USA, January 1977. doi:10.1145/512950.512973.

[CG02] Martine Ceberio and Laurent Granvilliers. Horner's rule for interval evaluation revisited. *Computing*, 69(1):51–81, 2002. doi:10.1007/s00607-002-1448-y.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. doi:10.1145/876638.876643.

[CGP17] Martin Clochard, Léon Gondelman, and Mário Pereira. The Matrix reproved. *Journal of Automated Reasoning*, 60(3):365–383, 2017. doi:10.1007/s10817-017-9436-2, hal:hal-01617437.

[CIJ+17] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in SMT. In V. Kuncak and R. Majumdar, editors, *29th International Conference on Computer Aided Verification (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 419–435, Heidelberg, Germany, July 2017. doi:10.1007/978-3-319-63390-9_22, hal:hal-01522770.

[CJL10] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *3rd International Congress on Mathematical Software (ICMS)*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Kobe, Japan, September 2010. doi:10.1007/978-3-642-15582-6_5.

[Cli90] William D. Clinger. How to read floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):92–101, 1990. doi:10.1145/93548.93557.

[CMRI12] Sylvain Conchon, Guillaume Melquiond, Cody Roux, and Mohamed Iguernelala. Built-in treatment of an axiomatic floating-point theory for SMT solvers. In *10th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 12–21, Manchester, UK, June 2012. hal:hal-01785166.

[CN08] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41(1):33–59, 2008. doi:10.1007/s10817-008-9101-x.

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

[CR87] George F. Corliss and Louis B. Rall. Adaptive, self-validating numerical quadrature. *SIAM Journal on Scientific and Statistical Computing*, 8(5):831–847, 1987. doi:10.1137/0908069.

[CS93] João Luiz Dihl Comba and Jorge Stolfi. Affine arithmetic and its applications to computer graphics. In *6th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pages 9–18, Recife, Brazil, 1993.

[CT95] Tim Coe and Ping Tak Peter Tang. It takes six ones to reach a flaw. In *12th IEEE Symposium on Computer Arithmetic*, pages 140–146, Bath, UK, July 1995. doi:10.1109/ARITH.1995.465365.

[CW80] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[D'A49] Jean le Rond D'Alembert. Recherches sur la courbe que forme une corde tendue mise en vibrations. In *Histoire de l'Académie Royale des Sciences et Belles Lettres (Année 1747)*, volume 3, pages 214–249. Haude et Spener, Berlin, 1749.

[dDLM06] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using Gappa. In *ACM Symposium on Applied Computing (SAC)*, pages 1318–1322, Dijon, France, 2006. URL: http://www.lri.fr/~melquion/doc/06-mcms-article.pdf.

[dDLM11] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011. doi:10.1109/TC.2010.128, hal:ensl-00200830.

[Dek71] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971. doi:10.1007/BF01397083.

[Del00] David Delahaye. A tactic language for the system Coq. In M. Parigot and A. Voronkov, editors, *7th International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95, La Réunion, France, November 2000. doi:10.1007/3-540-44404-1_7, hal:hal-01125070.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. doi:10.1145/360933.360975.

[DM04] Marc Daumas and Guillaume Melquiond. Generating formally certified bounds on values and round-off errors. In V. Brattka, C. Frougny, and N. Müller,

editors, *6th Conference on Real Numbers and Computers (RNC)*, pages 55–70, Schloß Dagstuhl, Germany, 2004. hal:inria-00070739.

[DM10] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):1–20, 2010. doi:10.1145/1644001.1644003, hal:hal-00127769.

[DMM05] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In P. Montuschi and E. Schwarz, editors, *17th IEEE Symposium on Computer Arithmetic (Arith)*, pages 188–195, Cape Cod, MA, USA, 2005. doi:10.1109/ARITH.2005.25, hal:hal-00164621.

[EM09] William Edmonson and Guillaume Melquiond. IEEE interval standard working group - P1788: current status. In J. D. Bruguera, M. Cornea, D. DasSarma, and J. Harrison, editors, *19th IEEE Symposium on Computer Arithmetic (Arith)*, pages 231–234, Portland, OR, USA, June 2009. doi:10.1109/ARITH.2009.36.

[FGK+00] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software: Practice and Experience*, 30(11):1167–1202, 2000. URL: https://www.cgal.org/, doi:10.1002/1097-024X(200009)30:11⟨1167::AID-SPE337⟩3.0.CO;2-B.

[Fig95] Samuel A. Figueroa. When is double rounding innocuous? *ACM SIGNUM Newsletter*, 30(3):21–26, 1995. doi:10.1145/221332.221334.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. doi:10.1007/978-3-540-73368-3_21, hal:inria-00270820.

[FMM17] Clément Fumex, Claude Marché, and Yannick Moy. Automated verification of floating-point computations in Ada programs. Research Report RR-9060, Inria Saclay–Île-de-France, April 2017. hal:hal-01511183.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In M. Felleisen and P. Gardner, editors, *22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128, Rome, Italy, March 2013. doi:10.1007/978-3-642-37036-6_8, hal:hal-00789533.

[Fu12] Bin Fu. Multivariate polynomial integration and differentiation are polynomial time inapproximable unless P=NP. In J. Snoeyink, P. Lu, K. Su, and L. Wang, editors, *Joint International Conference on Frontiers in Algorithmics and Algorithmic Aspects in Information and Management (FAW-AAIM)*, volume 7285 of *Lecture Notes in Computer Science*, pages 182–191, Beijing, China, May 2012. doi:10.1007/978-3-642-29700-7_17.

[GAC12] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. δ-complete decision procedures for satisfiability over the reals. In B. Gramlich, D. Miller, and U. Sattler, editors, *6th International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Artificial Intelligence*, pages 286–300, Manchester, UK, June 2012. doi:10.1007/978-3-642-31365-3_23.

[GM05] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 98–113, Oxford, UK, August 2005. doi:10.1007/11541868_7.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. doi:10.1145/103162.103163.

[GP06] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In K. Yi, editor, *International Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34, Seoul, Korea, August 2006. doi:10.1007/11823230_3.

[GT06] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In U. Furbach and N. Shankar, editors, *3rd International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lectures Notes in Artificial Intelligence*, pages 423–437, Seattle, WA, USA, August 2006. doi:10.1007/11814771_36.

[GW85] François Gramain and M. Weber. Computing an arithmetic constant related to the ring of Gaussian integers. *Mathematics of Computation*, 44(169):241–250, 1985. doi:10.1090/S0025-5718-1985-0771043-3.

[HAB+15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. 2015. arXiv:1501.02155.

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.

[Har97a] John Harrison. Floating-point verification in HOL light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[Har97b] John Harrison. Verifying the accuracy of polynomial approximations in HOL. In E. L. Gunter and A. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1275 of *Lecture Notes in Computer Science*, pages 137–152, Murray Hill, NJ, USA, August 1997. doi:10.1007/BFb0028391.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *12th International Conference in Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. doi:10.1007/3-540-48256-3_9.

[Har00a] John Harrison. Formal verification of floating-point trigonometric functions. In W. A. Hunt, Jr. and S. D. Johnson, editors, *3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233, Austin, TX, USA, November 2000. doi:10.1007/3-540-40922-X_14.

[Har00b] John Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 of *Lecture Notes in Computer Science*, pages 233–251, Portland, OR, USA, August 2000. doi:10.1007/3-540-44659-1_15.

[Har06] John Harrison. Floating-point verification using theorem proving. In M. Bernardo and A. Cimatti, editors, *6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*, volume 3965 of *Lecture Notes in Computer Science*, pages 211–242, Bertinoro, Italy, May 2006. doi:10.1007/11757283_8.

[Har07] John Harrison. Verifying nonlinear real formulas via sums of squares. In K. Schneider and J. Brandt, editors, *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany, September 2007. doi:10.1007/978-3-540-74591-4_9.

[Hel14] Harald A. Helfgott. Major arcs for Goldbach's problem. 2014. arXiv:1305.2897.

[Heu18] Marijn Heule. Schur number five. In *32nd AAAI Conference on Artificial Intelligence*, pages 6598–6606, New Orleans, LA, USA, February 2018. arXiv:1711.08076.

[HIH13] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *4th International Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *Lecture Notes in Computer Science*, pages 279–294, Rennes, France, July 2013. doi:10.1007/978-3-642-39634-2_21.

[Hol80] John E. Holm. *Floating-Point Arithmetic and Program Correctness Proofs*. PhD thesis, Cornell University, 1980. URL: https://ecommons.cornell.edu/handle/1813/6276.

[HS00] Joel Hass and Roger Schlafly. Double bubbles minimize. *Annals of Mathematics. Second Series*, 151(2):459–515, 2000. doi:10.2307/121042.

[IEE08] IEEE Computer Society. IEEE standard for floating-point arithmetic. Technical Report 754-2008, August 2008. doi:10.1109/IEEESTD.2008.4610935.

[IEE15] IEEE Computer Society. IEEE standard for interval arithmetic. Technical Report 1788-2015, June 2015. doi:10.1109/IEEESTD.2015.7140721.

[Imm18] Fabian Immler. A verified ODE solver and the Lorenz attractor. *Journal of Automated Reasoning*, 61(1):73–111, 2018. doi:10.1007/s10817-017-9448-y.

[IMN13] Daisuke Ishii, Guillaume Melquiond, and Shin Nakajima. Inductive verification of hybrid automata with strongest postcondition calculus. In E. B. Johnsen and L. Petre, editors, *10th Conference on Integrated Formal Methods (iFM)*, volume 7940 of *Lecture Notes in Computer Science*, pages 139–153, Turku, Finland, June 2013. doi:10.1007/978-3-642-38613-8_10, hal:hal-00806701.

[ISO11] ISO. International standard ISO/IEC 9899:2011, Programming languages – C, December 2011.

[IT19] Fabian Immler and Christoph Traut. The flow of ODEs: Formalization of variational equation and Poincaré map. *Journal of Automated Reasoning*, 62(2):215–236, 2019. doi:10.1007/s10817-018-9449-5.

[JJL12] Claude-Pierre Jeannerod and Jingyan Jourdan-Lu. Simultaneous floating-point sine and cosine for VLIW integer processors. In C. Silvano and R. Chamberlain, editors, *23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 69–76, Delft, Netherlands, July 2012. doi:10.1109/ASAP.2012.12.

[JKDW01] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Éric Walter. *Applied Interval Analysis. With Examples in Parameter and State Estimation, Robust Control and Robotics*. 379 pages. Springer-Verlag, 2001. doi:10.1007/978-1-4471-0249-6.

[JKMR11] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. *IEEE Transactions on Computers*, 60(2):214–227, 2011. doi:10.1109/TC.2010.152.

[Joh18] Fredrik Johansson. Numerical integration in arbitrary-precision ball arithmetic. In J. H. Davenport, M. Kauers, G. Labahn, and J. Urban, editors, *6th International Congress on Mathematical Software (ICMS)*, volume 10391 of *Lecture Notes in Computer Science*, pages 255–263, South Bend, IN, USA, July 2018. doi:10.1007/978-3-319-96418-8_30, hal:hal-01714969.

[Jol11] Mioara Joldeş. *Rigorous Polynomial Approximations and Applications*. PhD thesis, École Normale Supérieure de Lyon, France, 2011. hal:tel-00657843.

[Kar91] Ekaterina A. Karatsuba. Fast evaluation of transcendental functions. *Problemy Peredachi Informatsii*, 27(4):76–99, 1991.

[KL14] Olga Kupriianova and Christoph Q. Lauter. Metalibm: A mathematical functions code generator. In H. Hong and C. Yap, editors, *4th International Congress on Mathematical Software (ICMS)*, volume 8592 of *Lecture Notes in Computer Science*, pages 713–717, Seoul, South Korea, August 2014. doi:10.1007/978-3-662-44199-2_106.

[KMP+04] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. In S. Albers and T. Radzik, editors, *12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 702–713, Bergen, Norway, September 2004. doi:10.1007/978-3-540-30140-0_62.

[Knü94] Olaf Knüppel. PROFIL/BIAS—a fast interval library. *Computing*, 53(3):277–287, 1994. doi:10.1007/BF02307379.

[Knu98] Donald E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, MA, USA, 3rd edition, 1998.

[KO09] Cezary Kaliszyk and Russell O'Connor. Computing with classical real numbers. *Journal of Formalized Reasoning*, 2(1):27–39, 2009. doi:10.6092/issn.1972-5787/1411.

[Lag11] Jeffrey C. Lagarias. The Kepler conjecture and its proof. In *The Kepler Conjecture: The Hales-Ferguson Proof*, pages 3–26. 2011. doi:10.1007/978-1-4614-1129-1_1.

[Lee89] Corinna Lee. Multistep gradual rounding. *IEEE Transactions on Computers*, 38(4):595–600, 1989. doi:10.1109/12.21152.

[Lei13] K. Rustan M. Leino. Developing verified programs with Dafny. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering (ICSE)*, pages 1488–1490, San Francisco, CA, USA, May 2013. doi:10.1109/ICSE.2013.6606754.

[Lel13] Catherine Lelay. A new formalization of power series in Coq. In *5th Coq Workshop*, Rennes, France, July 2013. hal:hal-00880212.

[Lel15a] Catherine Lelay. How to express convergence for analysis in Coq. In *7th Coq Workshop*, Sophia Antipolis, France, June 2015. hal:hal-01169321.

[Lel15b] Catherine Lelay. *Repenser la bibliothèque réelle de Coq : vers une formalisation de l'analyse classique mieux adaptée*. PhD thesis, Université Paris Sud, 2015. hal:tel-01228517.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[Let02] Pierre Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *International Workshop on Types for Proofs and Programs (TYPES)*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219, Berg en Dal, Netherlands, April 2002. doi:10.1007/3-540-39185-1_12, hal:hal-00150914.

[LHD+10] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. Towards program optimization through automated analysis of numerical precision. In *8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 230–237, Toronto, ON, Canada, 2010. doi:10.1145/1772954.1772987.

[Lin76] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976. doi:10.1007/BF01931367.

[LM12] Catherine Lelay and Guillaume Melquiond. Différentiabilité et intégrabilité en Coq. Application à la formule de d'Alembert. In *23èmes Journées Francophones des Langages Applicatifs*, pages 119–133, Carnac, France, February 2012. hal:hal-00642206.

[LMRW14] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. Make it real: Effective floating-point reasoning via exact arithmetic. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4, Dresden, Germany, March 2014. doi:10.7873/DATE.2014.130.

[LSA18] Wonyeol Lee, Rahul Sharma, and Alex Aiken. On automatically proving the correctness of `math.h` implementations. In *45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 32 pages, Los Angeles, CA, USA, January 2018. doi:10.1145/3158135.

[May01] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, December 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*. 572 pages. Birkhäuser, 2010. doi:10.1007/978-0-8176-4705-6.

[MBdD+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 627 pages. Birkhäuser Basel, 2nd edition, 2018. doi:10.1007/978-3-319-76526-6.

[MCD17] Victor Magron, George Constantinides, and Alastair Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Transactions on Mathematical Software*, 43(4):34:1–34:31, 2017. URL: http://nl-certify.forge.ocamlcore.org/real2float.html, arXiv:1507.03331, doi:10.1145/3015465.

[MDM16] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016. doi:10.1007/s10817-015-9350-4, hal:hal-01086460.

[MDMM13] Érik Martin-Dorel, Guillaume Melquiond, and Jean-Michel Muller. Some issues related to double rounding. *BIT Numerical Mathematics*, 53(4):897–924, 2013. doi:10.1007/s10543-013-0436-2, hal:ensl-00644408.

[MDMP+13] Érik Martin-Dorel, Micaela Mayero, Ioana Pasca, Laurence Rideau, and Laurent Théry. Certified, efficient and sharp univariate taylor models in Coq. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 193–200, Timisoara, Romania, September 2013. doi:10.1109/SYNASC.2013.33, hal:hal-00845791.

[Mel06] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2006. hal:tel-01094485.

[Mel08a] Guillaume Melquiond. Floating-point arithmetic in the Coq system. In J. D. Bruguera and M. Daumas, editors, *8th Conference on Real Numbers and Computers (RNC)*, pages 93–102, Santiago de Compostela, Spain, July 2008. hal:hal-01780385.

[Mel08b] Guillaume Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, August 2008. doi:10.1007/978-3-540-71070-7_2, hal:hal-00180138.

[Mel12] Guillaume Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012. doi:10.1016/j.ic.2011.09.005, hal:hal-00797913.

[Mez10] Marc Mezzarobba. NumGfun: A package for numerical and analytic computation with D-finite functions. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 139–145, Munich, Germany, 2010. doi:10.1145/1837934.1837965.

[MG11] Niels Moller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2011. doi:10.1109/TC.2010.143.

[MK15] Claude Marché and Johannes Kanig. Bridging the gap between testing and formal verification in Ada development. *ERCIM News*, 100:38–39, January 2015. URL: https://ercim-news.ercim.eu/en100/r-i/bridging-the-gap-between-testing-and-formal-verification-in-ada-development.

[MLK98] J. Strother Moore, Tom W. Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, 1998. doi:10.1109/12.713311.

[MMSP16] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. In J. C. Blanchette and S. Merz, editors, *7th Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *Lecture Notes in Computer Science*, pages 274–289, Nancy, France, August 2016. doi:10.1007/978-3-319-43144-4_17, hal:hal-01289616.

[MMSP19] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *Journal of Automated Reasoning*, 62(2):281–300, 2019. doi:10.1007/s10817-018-9463-7, hal:hal-01630143.

[MN13] César Muñoz and Anthony Narkawicz. Formalization of Bernstein polynomials and applications to global optimization. *Journal of Automated Reasoning*, 51(2):151–196, 2013. doi:10.1007/s10817-012-9256-3.

[MNZ13] Guillaume Melquiond, W. Georg Nowak, and Paul Zimmermann. Numerical approximation of the Masser-Gramain constant to four decimal digits: $\delta = 1.819...$. *Mathematics of Computation*, 82:1235–1246, 2013. doi:10.1090/S0025-5718-2012-02635-4, hal:hal-00644166.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3):1–41, May 2008. doi:10.1145/1353445.1353446.

[Moo63] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1963.

[MP05] Guillaume Melquiond and Sylvain Pion. Formal certification of arithmetic filters for geometric predicates. In *17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005. hal:inria-00344518.

[MP07] Guillaume Melquiond and Sylvain Pion. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*, 41(1):57–70, 2007. doi:10.1051/ita:2007005, hal:inria-00071232.

[MP09] Guillaume Melquiond and Sylvain Pion. Directed rounding arithmetic operations. Technical Report 2899, ISO C++ Standardization Committee, 2009. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2899.pdf.

[MR11] Christophe Mouilleron and Guillaume Revy. Automatic generation of fast and certified code for polynomial evaluation. In E. Antelo, D. Hough, and P. Ienne, editors, *20th IEEE Symposium on Computer Arithmetic*, pages 233–242, Tübingen, Germany, July 2011. doi:10.1109/ARITH.2011.39.

[MRH18] Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 framework for reflection proofs and its application to GMP's algorithms. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *9th International Joint Conference on Automated Reasoning (IJCAR)*, volume 10900 of *Lecture Notes in Artificial Intelligence*, pages 178–193, Oxford, United Kingdom, July 2018. doi:10.1007/978-3-319-94205-6_13, hal:hal-01699754.

[MS13] Evgeny Makarov and Bas Spitters. The Picard algorithm for ordinary differential equations in Coq. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *4th International Conference on Interactive Theorem Proving (ITP)*, volume 7998 of *Lecture Notes in Computer Science*, pages 463–468, Rennes, France, July 2013. doi:10.1007/978-3-642-39634-2_34.

[Mul16] Jean-Michel Muller. *Elementary Functions. Algorithms and Implementation*. 283 pages. Birkhäuser, Boston, MA, 3rd edition, 2016. doi:10.1007/978-1-4899-7983-4.

[Ned06] Nedialko S. Nedialkov. Interval tools for ODEs and DAEs. In *12th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN)*, 12 pages, Duisburg, Germany, 2006. URL: http://www.cas.mcmaster.ca/~nedialk/vnodelp/, doi:10.1109/SCAN.2006.28.

[OBO+16] Katsuhisa Ozaki, Florian Bünger, Takeshi Ogita, Shin'ichi Oishi, and Siegfried M. Rump. Simple floating-point filters for the two-dimensional orientation problem. *BIT Numerical Mathematics*, 56(2):729–749, 2016. doi:10.1007/s10543-015-0574-9.

[Pra95] Vaughan Pratt. Anatomy of the Pentium bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *6th International Joint Conference CAAP/FASE, Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. doi:10.1007/3-540-59293-8_189.

[Pri91] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *10th IEEE Symposium on Computer Arithmetic*, pages 132–143, Grenoble, France, June 1991. doi:10.1109/ARITH.1991.145549.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conference on Supercomputing*, pages 4–13, Albuquerque, New Mexico, United States, 1991. doi:10.1145/125826.125848.

[Rao17] Michaël Rao. Exhaustive search of convex pentagons which tile the plane. 2017. arXiv:1708.00274.

[RHMM17] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In A. Paskevich and T. Wies, editors, *9th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017. doi:10.1007/978-3-319-72308-2_6, hal:hal-01519732.

[Rou14] Pierre Roux. Innocuous double rounding of basic arithmetic operations. *Journal of Formalized Reasoning*, 7(1):131–142, 2014. doi:10.6092/issn.1972-5787/4359.

[RSSvdH96] Daniel Richardson, Bruno Salvy, John Shackell, and Joris van der Hoeven. Asymptotic expansions of exp-log functions. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 309–313, Zurich, Switzerland, July 1996. doi:10.1145/236869.237089.

[Rum99] Siegfried M. Rump. INTLAB — INTerval LABoratory. In T. Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, 1999. URL: http://www.ti3.tu-harburg.de/rump/intlab/, doi:10.1007/978-94-017-1247-7_7.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. doi:10.1112/S1461157000000176.

[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):75–125, 1999. doi:10.1023/A:1008669628911.

[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. In W. A. Hunt, Jr. and S. D. Johnson, editors, *3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954, pages 3–36, November 2000. doi:10.1007/3-540-40922-X_3.

[RZBM09] Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT Numerical Mathematics*, 49(2):419–431, 2009. doi:10.1007/s10543-009-0218-z, hal:inria-00337537.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. 484 pages. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons, 1998.

[SJRG15] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In N. Bjørner and F. de Boer, editors, *20th International Symposium on Formal Methods (FM)*, volume 9109 of *Lecture Notes in Programming and Software Engineering*, pages 532–555, Oslo, Norway, June 2015. doi:10.1007/978-3-319-19249-9_33.

[Ske92] Robert Skeel. Roundoff error cripples Patriot missile. *SIAM News*, 25(4):11, 1992. URL: http://www-users.math.umn.edu/~arnold/disasters/Patriot-dharan-skeel-siam.pdf.

[Spe80] Bert Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. PhD thesis, Illinois University, Urbana, 1980. doi:10.2172/5254402.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144, Austin, TX, USA, November 2000. doi:10.1007/3-540-40922-X_8.

[Sta95] Volker Stahl. *Interval Methods for Bounding the Range of Polynomials And Solving Systems of Nonlinear Equations*. PhD thesis, Johannes Kepler Universität, Linz, September 1995.

[Ste74] Pat H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[TMM+18] Laura Titolo, Mariano M. Moscato, César A. Muñoz, Aaron Dutle, and François Bobot. A formally verified floating-point implementation of the compact position reporting algorithm. In K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, editors, *22nd International Symposium on Formal Methods (FM)*, volume 10951

of *Lecture Notes in Programming and Software Engineering*, pages 364–381, Oxford, UK, July 2018. doi:10.1007/978-3-319-95582-7_22.

[vN45] John von Neumann. First draft of a report on the EDVAC. June 1945.

[YD95] Chee K. Yap and Thomas Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, 1995. doi:10.1142/9789812831699_0011.

[Ziv91] Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, 1991. doi:10.1145/114697.116813.

[ZJLS01] Jun Zhang, Karl H. Johansson, John Lygeros, and Shankar Sastry. Zeno hybrid systems. *International Journal of Robust and Nonlinear Control*, 11(5):435–451, 2001. doi:10.1002/rnc.592.

[Zum06] Roland Zumkeller. Formal global optimisation with Taylor models. In U. Furbach and N. Shankar, editors, *3th International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 408–422, Seattle, WA, USA, August 2006. doi:10.1007/11814771_35.

Index